Universität Ulm
Abt. Datenbanken und Informationssysteme
Leiter: Prof. Dr. Peter Dadam

Workflow- und Prozeßsynchronisation mit Interaktionsausdrücken und -graphen

Konzeption und Realisierung eines Formalismus zur Spezifikation und Implementierung von Synchronisationsbedingungen

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Informatik
der Universität Ulm

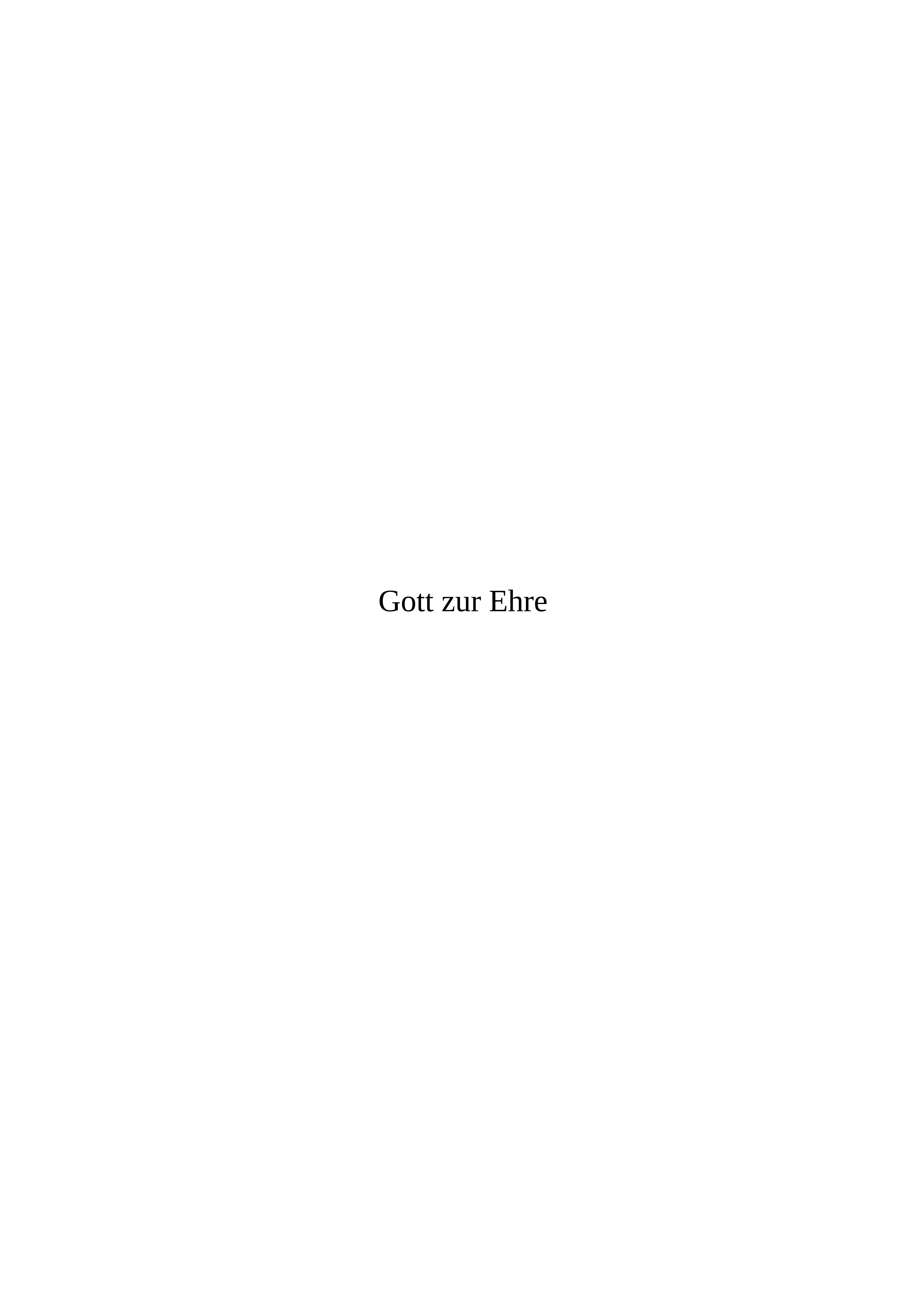
vorgelegt von
Christian Heinlein
aus Ulm

Mai 2000

Amtierender Dekan: Prof. Dr. Uwe Schöning

Gutachter: Prof. Dr. Peter Dadam
Prof. Dr. Helmuth Partsch
Prof. Dr. Walter Vogler (Universität Augsburg)

Tag der Promotion: 20. Juli 2000


Gott zur Ehre

Vorwort und Dank

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Abteilung Datenbanken und Informationssysteme der Universität Ulm. Obwohl ihre Wurzeln zweifellos in den Projekten OKIS (Offenes klinisches Datenbank- und Informationssystem zur Integration autonomer Subsysteme) und ADEPT (Application Development Based on Premodeled Activity Templates) liegen, stellt sie doch eine eigenständige, in sich geschlossene Arbeit dar, deren Resultate nicht nur auf die ursprüngliche Aufgabenstellung „Handhabung von Inter-Workflow-Abhängigkeiten", sondern auch auf andere Problemstellungen im Bereich der Prozeßsynchronisation angewandt werden können. Auch das immer wieder aufgegriffene Szenario der medizinischen Untersuchungsworkflows sollte lediglich als ein Musterbeispiel zur Motivation und Illustration, und nicht etwa als ausschließliches Anwendungsfeld der entwickelten Konzepte verstanden werden.

Mein Dank gilt zunächst meinem Doktorvater, Herrn Prof. Dr. Peter Dadam, dem es im Laufe der Jahre gelungen ist, aus einem wissenschaftlich naiven Tüftler einen gelegentlich immer noch gerne tüftelnden Wissenschaftler zu machen. Danken möchte ich weiterhin den beiden anderen Gutachtern, Herrn Prof. Dr. Helmuth Partsch und Herrn Prof. Dr. Walter Vogler (Augsburg), deren konstruktive Kritik und detaillierte Rückmeldung wesentlich zur Verbesserung der Arbeit beigetragen hat. Allen Kolleginnen und Kollegen der Abteilung danke ich herzlich für die gute Zusammenarbeit, für manche fruchtbaren Diskussionen und ihre tatkräftige Unterstützung beim Korrekturlesen. Nicht vergessen möchte ich meinen derzeitigen Projektleiter, Herrn Prof. Dr. Wolfgang Klas, der mir am Schluß den nötigen Freiraum zur Fertigstellung der Arbeit einräumte.

Meinen Eltern danke ich ganz herzlich für ihre Liebe und für die selbstverständliche Unterstützung meines Werdegangs. Meinen Kindern Matthias und Manuela gebührt großes Lob für ihr Verständnis und ihre Geduld, wenn es immer wieder hieß: „Papa muß Doktorarbeit schreiben." Meine „Brüder und Schwestern im Herrn" trugen mich durch ihre Gebete und ihren Glauben.

Meine Frau Antje müßte eigentlich als Koautorin genannt werden, denn ohne ihre fortwährende Liebe, Ermutigung, Unterstützung und Entlastung wäre diese Arbeit wohl nie fertig geworden.

Der größte Dank gebührt jedoch meinem himmlischen Vater, der mich „erzogen hat, wie ein Mann seinen Sohn erzieht" (5. Mose 8:5), der „ein Schild um mich her war" (Psalm 3:4), wenn Zweifel mich bedrohten, und „durch dessen Gnade ich bin, was ich bin" (1. Korinther 15:10).

SOLI DEO GLORIA

Günzburg, im Mai 2000 *Christian Heinlein*

Kurzgliederung

Inhaltsverzeichnis

Kapitel 1

Motivation, Aufgabenstellung und Überblick

1.1 Workflow-Management

1.1.1 Workflows

Ein *Workflow* ist prinzipiell ein beliebiger Arbeitsablauf, der immer wieder nach demselben − oder zumindest einem ähnlichen − Schema abläuft. Er besteht aus einzelnen *Arbeitsschritten*, die meist von unterschiedlichen *Bearbeitern* ausgeführt werden, denen hierfür geeignete Anwendungsprogramme, sogenannte *Schrittprogramme*, zur Verfügung stehen [Jablonski95ab, Georgakopoulos95, Jablonski97]. Typische Workflows sind z. B.

- die Bearbeitung eines Schadensfalls in einer Versicherung [Georgakopoulos95],
- die Vergabe eines Kredits in einer Bank [Alonso95],
- die Vorbereitung, Durchführung und Abrechnung einer Dienstreise [Vogel92, Jablonski95ab],
- die Untersuchung eines Patienten in einem Krankenhaus, einschließlich aller hierfür erforderlichen Vor- und Nachbereitungen [Dadam95, Kuhn95ab, Konyen96b, Haimowitz96, Reichert97a] (vgl. auch Abb. 1.1 und 1.2).

1.1.2 Workflow-Management-Systeme

Ein *Workflow-Management-System* (WfMS) ist ein Software-System, dessen zentrale Aufgabe darin besteht, für die korrekte *Ausführung* von Workflows zu sorgen [Jablonski97]. Zu diesem Zweck wird für jeden Bearbeiter eine *Arbeitsliste* verwaltet, in die alle Arbeitsschritte eingetragen werden, die von diesem Bearbeiter auszuführen sind. Sobald ein Arbeitsschritt beendet ist, sorgt das WfMS (bzw. dessen zentrale *Ausführungseinheit*, vgl. § 1.1.6) dafür, daß der Workflow zum Bearbeiter des nächsten Schritts „weitertransportiert“ wird und in dessen Arbeitsliste erscheint. Auf diese Weise werden zum einen die einzelnen Bearbeiter entlastet, weil sie sich nicht mehr um die korrekte (und möglichst schnelle) Weiterleitung von Akten o. ä. kümmern müssen, zum anderen bietet das WfMS autorisierten Benutzern die Möglichkeit, jederzeit den *momentanen Bearbeitungsstand* eines Arbeitsablaufs einzusehen, was z. B. zur Beantwortung von telefonischen Kundenanfragen sehr hilfreich sein kann.

In der Regel gibt es für einen Arbeitsschritt nicht nur einen, sondern eine ganze *Menge* potentieller Bearbeiter, die diesen Schritt ausführen können, weil sie entsprechende Fähigkeiten oder Kompetenzen besitzen, d. h. weil sie die erforderliche *Rolle* oder *Funktion* einnehmen können. In diesem Fall wird der Schritt zunächst allen in Frage kommenden Bearbeitern zur Ausführung angeboten; sobald einer von ihnen den Schritt tatsächlich ausführt (oder für sich reserviert, d. h. erklärt, daß er ihn später ausführen wird), sorgt das WfMS dafür, daß er aus den Arbeitslisten der übrigen Bearbeiter wieder entfernt wird [Jablonski95ab].[1]

1.1.3 Grundprinzip von Workflow-Management

Im Gegensatz zu „monolithischen“ Anwendungssystemen, bei denen Arbeitsabläufe meist fest im Programmcode „verdrahtet“ sind, besteht ein wichtiges *Grundprinzip von Workflow-Management* darin, daß der *Kontroll- und Datenfluß* eines Workflows *explizit* und *unabhängig* von der Implementie-

[1] Je nach System erfolgt diese Aktualisierung der Arbeitslisten entweder vollautomatisch oder nur auf explizite Anforderung eines Bearbeiters. In jedem Fall wird jedoch sichergestellt, daß ein Schritt von genau einem Bearbeiter ausgeführt wird.

rung der einzelnen Schrittprogramme festgelegt und somit auch geändert werden kann [Jablonski97]. Auf diese Weise ist es möglich, Arbeitsabläufe schnell und ohne große Mühe an sich ändernde Gegebenheiten *anzupassen*, weil i. d. R. nur die Spezifikation des Gesamtablaufs, nicht jedoch der Programmcode einzelner Anwendungen geändert werden muß. Umgekehrt ist es auch möglich, die Funktionalität einzelner Anwendungsprogramme zu modifizieren oder Programme wie „Steckmodule" auszutauschen, ohne daß hiervon die Struktur des Gesamtablaufs betroffen ist [Georgakopoulos95].

1.1.4 Modellierung von Workflows

Zur *Modellierung* von Workflows, d. h. zur Spezifikation der einzelnen Arbeitsschritte (einschließlich zugehöriger Bearbeiter/Rollen und Schrittprogramme) sowie zur Festlegung des Kontroll- und Datenflusses zwischen den Schritten, werden meist graphische Editoren verwendet, wobei als Kontrollkonstrukte neben den aus imperativen Programmiersprachen bekannten Operatoren Sequenz, Verzweigung und Wiederholung häufig auch Konstrukte zur Formulierung paralleler „Zweige" angeboten werden [Jablonski95ab, Georgakopoulos95, Jablonski97]. Da im Kontext dieser Arbeit primär der Kontrollfluß von Workflows relevant ist, wird auf die Spezifikation der übrigen Aspekte (Datenflußmodellierung, Zuordnung von Bearbeitern und Schrittprogrammen usw.) nicht näher eingegangen.

1.1.5 Beispiel

Die Abbildungen 1.1 und 1.2 zeigen zwei verschiedene Varianten des Arbeitsablaufs „Medizinische Untersuchung" [Kuhn95ab, Konyen96b, Reichert97a].

Nach der *Anordnung* einer bestimmten Untersuchung (wie z. B. einer Sonographie oder einer Endoskopie) durch den Stationsarzt, muß zunächst mit der leistungserbringenden Stelle (z. B. Innere Medizin) ein *Termin vereinbart* werden. Gleichzeitig kann der Patient für die Untersuchung *vorbereitet* und, falls erforderlich, über mögliche Risiken und Nebenwirkungen *aufgeklärt* werden. Nach Abschluß dieser Vorbereitungen kann der Patient zur Untersuchung *abgerufen* und die eigentliche Untersuchung *durchgeführt* werden. Im Anschluß daran erstellt der untersuchende Arzt einen *Befund*, der vom Stationsarzt, der die Untersuchung angeordnet hatte, *gelesen* wird. Je nach Untersuchungsart, wird neben einem sofort erstellten *Kurzbefund* zusätzlich ein ausführlicher *Langbefund* erstellt.

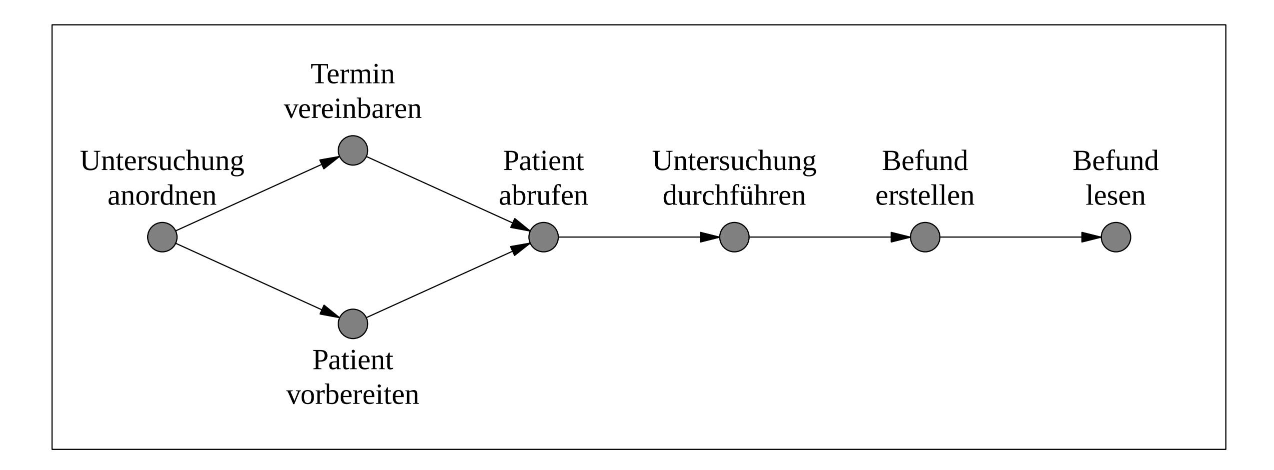


Abbildung 1.1: Workflow „Sonographische Untersuchung"

Die Pfeile zwischen den einzelnen Arbeitsschritten (die als Kreise dargestellt sind) entsprechen *Kontrollflußkanten*, die die Ausführungsreihenfolge der Schritte im Sinne einer partiellen Ordnung festlegen: Ein bestimmter Schritt darf ausgeführt werden, sobald alle seine *Vorgängerschritte* beendet

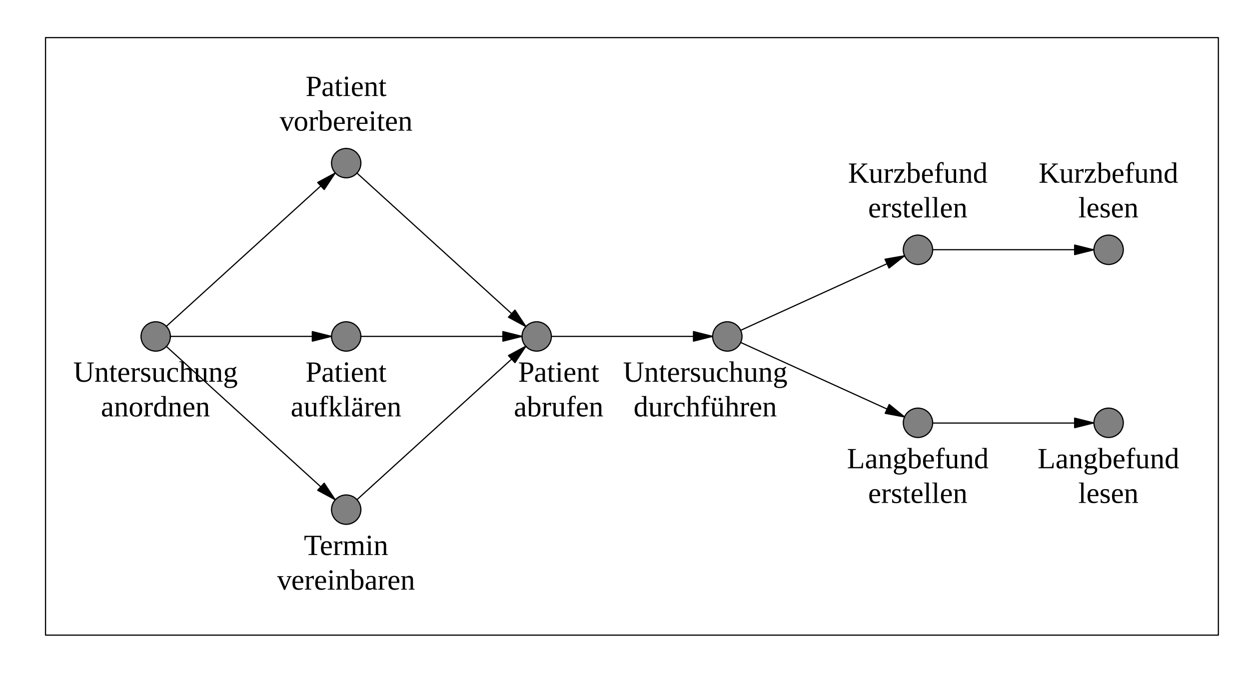


Abbildung 1.2: Workflow „Endoskopische Untersuchung“

sind. Beispielsweise können die Schritte Patient vorbereiten, Patient aufklären und Termin vereinbaren in Abb. 1.2 nach Belieben parallel ausgeführt werden, sobald Untersuchung anordnen beendet ist; Patient abrufen kann nach Beendigung dieser drei Schritte ausgeführt werden usw.

1.1.6 Referenzmodell der Workflow Management Coalition

Die *Workflow Management Coalition* (WfMC), ein internationales Gremium von Herstellern und Anwendern von Workflow-Management-Systemen [WfMC96, Versteegen95], hat im Rahmen ihrer Standardisierungsbemühungen ein *Referenzmodell* für Workflow-Management-Systeme entworfen [Hollingsworth94], das in Abb. 1.3 etwas vereinfacht dargestellt ist. Demnach besteht ein WfMS im engeren Sinn aus einer zentralen *Ausführungseinheit* (engl. workflow engine), deren Aufgaben in § 1.1.2 skizziert wurden und die über verschiedene *Schnittstellen* (numeriert von 1 bis 5) mit den folgenden Arten von Programmen (und Personen) interagieren kann:

1. *Workflow-Editoren* zur graphischen Modellierung von Workflows durch *Workflow-Modellierer*;
2. *Arbeitslistenprogramme* (engl. worklist handler) als Schnittstelle zu den einzelnen *Bearbeitern*;
3. *Schrittprogramme* zur Ausführung einzelner Arbeitsschritte durch *Bearbeiter*;
4. ggf. andere *Workflow-Ausführungseinheiten* zur wechselseitigen Delegation der Ausführung von (Teil-)Workflows;
5. *Verwaltungsprogramme* zur Modellierung von Organisationsstrukturen (wie z. B. Abteilungen, Mitarbeiter, Rollen, Vertreterregelungen etc.) sowie zur „Inspektion“ des Bearbeitungsstands laufender Workflows durch *Systemverwalter* (oder andere autorisierte Personen).

Ein WfMS im weiteren Sinn umfaßt neben der zentralen Ausführungseinheit normalerweise auch Editoren zur Modellierung von Workflows und Organisationsstrukturen (die oft auch als *Buildtime*-Komponenten des Systems bezeichnet werden) sowie vordefinierte Arbeitslistenprogramme, die bei Bedarf jedoch durch spezialisierte Anwendungen ersetzt werden können (und zusammen mit der Ausführungseinheit auch als *Runtime*-Komponenten des Systems bezeichnet werden).

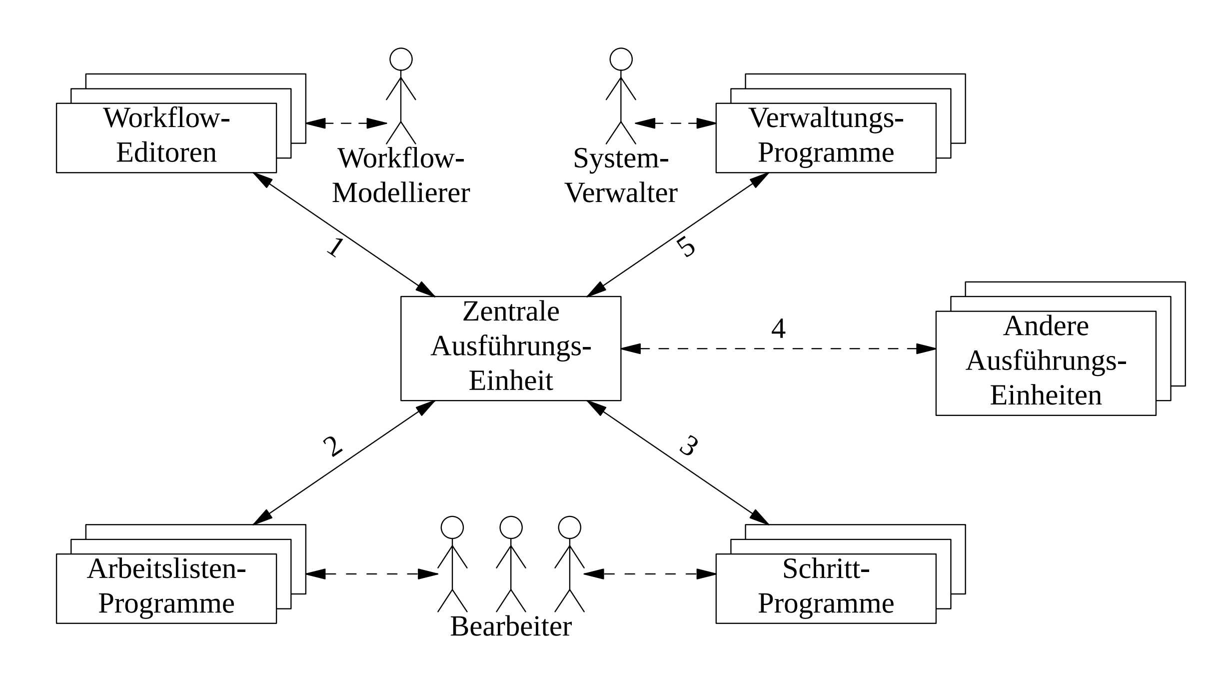


Abbildung 1.3: Referenzmodell der Workflow Management Coalition

1.2 Inter-Workflow-Abhängigkeiten

1.2.1 Parallele Workflows

Ähnlich wie Prozesse in einem Betriebssystem, können Workflows natürlich auch *parallel* ausgeführt werden. Allerdings bieten heutige Workflow-Management-Systeme − unabhängig davon, ob es sich um kommerzielle Produkte oder um Forschungsprototypen handelt − in der Regel keine Möglichkeiten, um parallel laufende Workflows zu *synchronisieren*; sie werden im wesentlichen *unabhängig* voneinander ausgeführt (vgl. auch § 6.6.2).

Solange sich zwei parallel laufende Untersuchungsworkflows im vorangegangenen Beispiel auf *verschiedene* Patienten beziehen, stellt dies auch kein Problem dar; in diesem Fall dürfen sich die Ausführungen der beiden Workflows beliebig überlappen. Sobald sich die Untersuchungen jedoch auf *denselben* Patienten beziehen − was in der Praxis häufig der Fall ist, weil für einen Patienten oft mehrere Untersuchungen auf einmal angeordnet werden −, müssen die beiden Workflows teilweise synchronisiert werden [Kuhn95ab, Reichert97a].

1.2.2 Workflow-übergreifende Synchronisationsbedingungen

So wäre es z. B. sinnvoll, die Vorbereitungsmaßnahmen beider Untersuchungen *zusammenzufassen* − einerseits, um die Arbeit des medizinischen Personals zu reduzieren, und andererseits, um unnötige Belastungen des Patienten zu vermeiden. Wenn beispielsweise im Rahmen der Vorbereitung jeweils eine Blutprobe entnommen werden muß, so hätte ein Patient sicherlich kein Verständnis dafür, wenn er sich dieser unangenehmen Prozedur zweimal unterziehen müßte, obwohl eine einzige Probe für beide Untersuchungen zusammen genügen würde.

In gewisser Weise umgekehrt verhält es sich mit der Folge der Schritte Patient abrufen und Untersuchung durchführen: Da sich ein Patient natürlich nicht in zwei Untersuchungsstellen gleichzeitig befinden kann, müssen diese Schritte − auch über Workflowgrenzen hinweg − strikt sequentiell ausgeführt werden. Heutzutage werden derartige Synchronisationsprobleme meist per Telefon „gelöst“: Wenn ein Patient, der sich gerade nicht auf der Station befindet, zu einer Untersuchung gerufen wird,

informiert die Stationsschwester die entsprechende Untersuchungsstelle und bittet sie, einen anderen Patienten vorzuziehen. Beim Einsatz eines Workflow-Management-Systems wäre es jedoch wünschenswert, wenn Arbeitsschritte (wie z. B. Patient abrufen), die zwar aus Sicht eines einzelnen Workflows (und damit aus Sicht des WfMSs) prinzipiell zulässig sind, deren Ausführung jedoch gegen eine solche *Workflow-übergreifende* Synchronisationsbedingung verstoßen würde, aus den Arbeitslisten der Benutzer (vorübergehend) entfernt werden, damit sie auch tatsächlich nicht ausgeführt werden können.[2]

1.2.3 Aufgabenstellung

Aus dieser Anforderung ergibt sich unmittelbar die *Aufgabenstellung* für die vorliegende Arbeit: Zum einen wird ein *Formalismus* benötigt, mit dessen Hilfe derartige *Inter-Workflow-Abhängigkeiten* sinnvoll beschrieben werden können; zum anderen muß sichergestellt werden, daß die so beschriebenen Synchronisations- oder *Integritätsbedingungen* zur Laufzeit auch eingehalten werden, d. h. daß ein Arbeitsschritt nur zur Ausführung angeboten wird, wenn durch seine Ausführung keine Bedingungen verletzt werden.

Bevor dies in § 1.4 näher erläutert wird, soll im folgenden − anhand einiger *nicht* zum gewünschten Ziel führender Lösungsversuche − verdeutlicht werden, daß das Problem der Inter-Workflow-Abhängigkeiten *real* existiert (d. h. nicht einfach „wegdefiniert“ werden kann) und mit heutiger Workflow-Management-Technologie *nicht* zufriedenstellend gelöst werden kann.

1.3 Lösungsversuche

1.3.1 Verschmelzung abhängiger Workflows

1.3.1.1 Ansatz

Wenn zwischen zwei (oder mehreren) Workflows Abhängigkeiten bestehen, so ist die Frage berechtigt, ob diese möglicherweise auf eine ungeschickte Modellierung der Workflows zurückzuführen sind, bei der logisch zusammengehörende Arbeitsschritte fälschlicherweise auf mehrere Workflows verteilt wurden. Oder anders ausgedrückt: Kann man Workflows, zwischen denen Abhängigkeitsbeziehungen bestehen, nicht einfach zu einem einzigen Workflow *verschmelzen* und auf diese Weise die vorhandenen *Inter*-Workflow-Abhängigkeiten auf gewöhnliche *Intra*-Workflow-Abhängigkeiten (d. h. Abhängigkeiten zwischen Schritten eines *einzigen* Workflows) zurückführen, die mit Hilfe der üblichen Kontrollkonstrukte beschrieben werden können?

Als Beispiel betrachte man den Workflow in Abb. 1.4, der durch eine Verschmelzung der beiden Workflows aus Abb. 1.1 und 1.2 entstanden ist, und der beschreibt, welche Arbeitsschritte für einen Patienten auszuführen sind, für den sowohl eine Sonographie als auch eine Endoskopie durchgeführt werden soll. Man beachte, daß der Schritt Patient vorbereiten nur noch einmal vorhanden ist, und daß die Sonographie *vor* der Endoskopie durchgeführt werden muß, was in diesem speziellen Fall auch medizinisch sinnvoll ist, weil eine Endoskopie die Ergebnisse einer nachfolgenden Sonographie möglicherweise verfälschen würde.[3]

1.3.1.2 Kritik

Für dieses einfache Beispiel scheint der Ansatz der Workflow-Verschmelzung durchaus sinnvoll zu sein. Wenn man jedoch bedenkt, daß man in realistischen Anwendungen sicher nicht nur zwei, sondern mindestens fünf bis zehn voneinander abhängige Workflows vorliegen hat,[4] von denen bereits je-

[2] Alternativ könnte man sich vorstellen, daß derartige Schritte in den Arbeitslisten lediglich „deaktiviert“ werden, so daß ein Bearbeiter erkennen kann, daß sie prinzipiell zur Ausführung anstehen, momentan aber nicht ausgeführt werden dürfen.
[3] Für andere Kombinationen von Untersuchungsarten lassen sich solche Reihenfolgen jedoch i. d. R. nicht a priori festlegen.
[4] Die Anzahl der verschiedenen Untersuchungsarten in einer internistischen Klinik liegt z. B. mindestens in dieser Größenordnung.

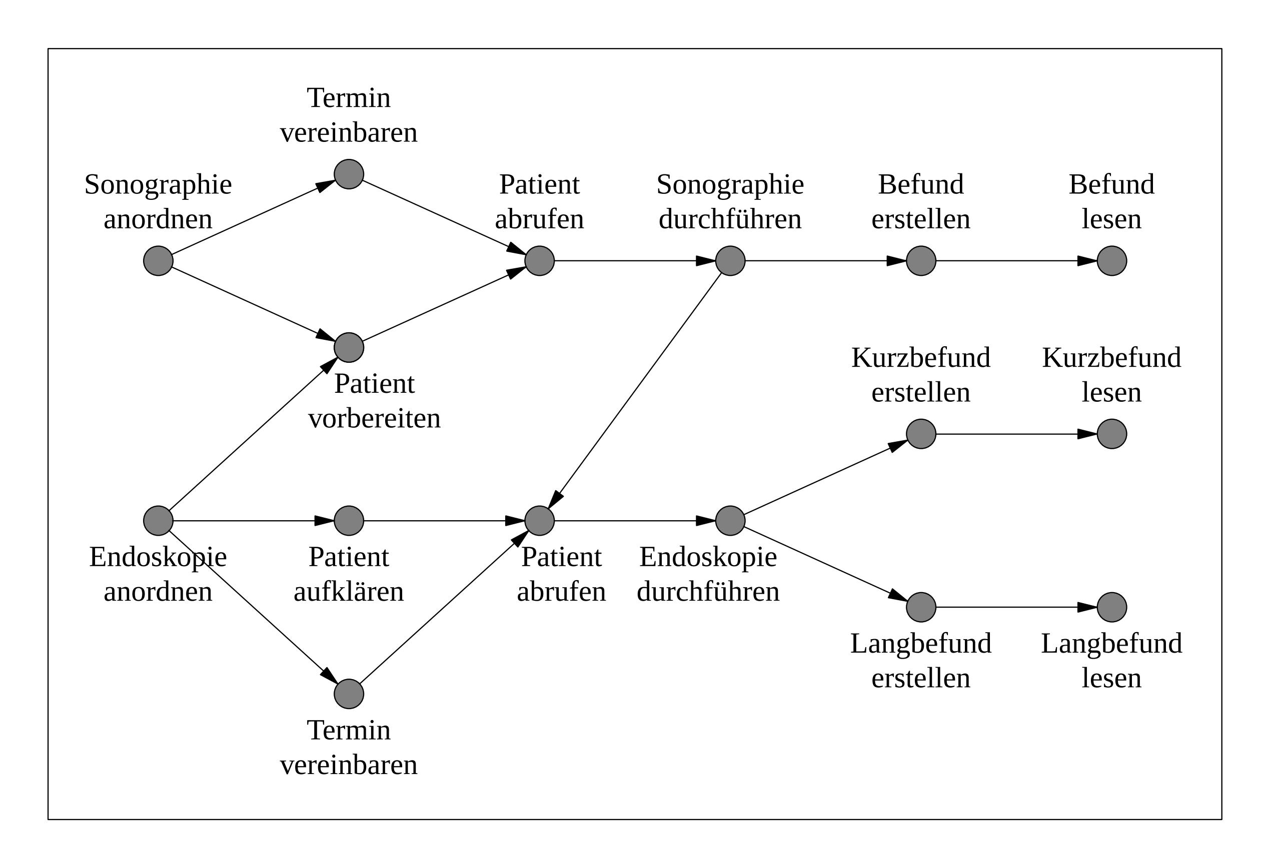


Abbildung 1.4: Verschmelzung von Workflows

der einzelne um einiges komplexer ist als die hier gezeigten, vereinfachten Beispiele [Schultheiß95ab, Schultheiß96, Konyen96abc], so wird schnell klar, daß dieser Vorgehensweise durch die *Komplexität* der resultierenden Gesamtworkflows enge Grenzen gesetzt sind. Da sich Abhängigkeitsbeziehungen außerdem transitiv fortpflanzen, ist die Wahrscheinlichkeit relativ groß, daß man bei einer konsequenten Anwendung dieser Methode im Endeffekt *sämtliche* Workflows einer Klinik (oder eines anderen Unternehmens) zu einem einzigen „Mega-Workflow" verschmelzen müßte, dessen Komplexität jedes beherrschbare Maß übersteigt.

Aus diesen Gründen muß die auf den ersten Blick durchaus vernünftig erscheinende Idee, Inter-Workflow-Abhängigkeiten durch das Verschmelzen abhängiger Workflows zu eliminieren, als nicht praktikabel eingestuft werden.

1.3.2 Explizite Synchronisierung abhängiger Workflows

1.3.2.1 Ansatz

Da das Problem der Inter-Workflow-Abhängigkeiten also, wie erläutert, nicht einfach „wegdefiniert" werden kann, wird als nächstes versucht, abhängige Workflows *explizit* und *individuell* zu synchronisieren. Einige wenige Workflow-Management-Systeme (wie z. B. SNI WorkParty [Rupietta94, SNI95, Frank97] oder SAP BusinessFlow [Wächter95ab, Gerstner95]) bieten hierfür ein rudimentäres *Ereignis-Konzept* an, mit dessen Hilfe sich Workflows gegenseitig Ereignisse oder *Nachrichten* senden können. Außerdem kann bei der Modellierung eines Workflows festgelegt werden, daß ein bestimmter Schritt dieses Workflows erst ausgeführt werden darf, nachdem eine bestimmte Nachricht von einem *anderen* Workflow eingetroffen ist.

Konzeptionell hat man damit die Möglichkeit, Kontrollflußkanten *über Workflowgrenzen hinweg* zu spezifizieren, wie dies in Abb. 1.5 schematisch dargestellt ist. Die beiden Workflows für Sonographie und Endoskopie sind dort wieder (wie ursprünglich in Abb. 1.1 und 1.2) getrennt, wurden jedoch um

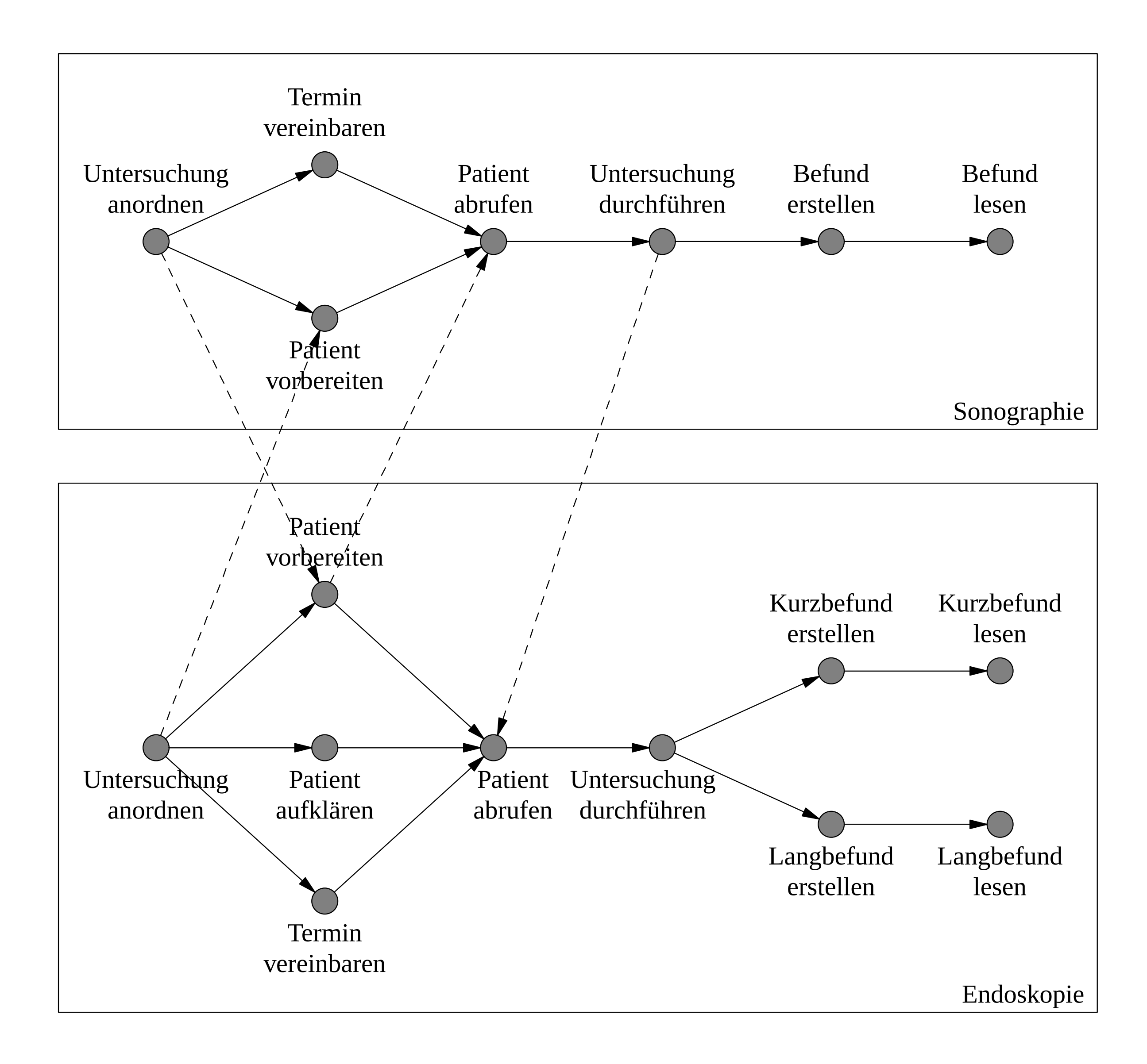


Abbildung 1.5: Explizite Synchronisierung abhängiger Workflows

einige „Send-“ und „Receive-Anweisungen“ erweitert, die durch die workflowübergreifenden, gestrichelten Pfeile dargestellt sind. Durch die wechselseitige Verknüpfung von Untersuchung anordnen des einen Workflows mit Patient vorbereiten des jeweils anderen Workflows soll erreicht werden, daß die beiden Vorbereitungsschritte *gleichzeitig* zur Ausführung angeboten werden und daher faktisch verschmolzen werden können, auch wenn es sich formal um zwei getrennte Schritte handelt. Anschließend kann der Patient, wie in Abb. 1.4, zur Sonographie, und nach deren Durchführung zur Endoskopie gerufen werden.

1.3.2.2 Kritik

Diese Vorgehensweise hat zwar gegenüber der Verschmelzung abhängiger Workflows den Vorteil, daß einzelne Workflows separat modelliert werden können und ihre *Größe* dadurch rein äußerlich überschaubar bleibt. An der inhärenten *Komplexität* der Definitionen ändert diese Tatsache jedoch nur wenig, da man zum Verständnis einer einzelnen Workflow-Spezifikation nun u. U. sämtliche direkt oder indirekt mit diesem Workflow „verbundenen“ Definitionen benötigt. Hinzu kommt, daß man bei der Modellierung einzelner Workflows die workflowübergreifenden Pfeile aus Abb. 1.5 gar nicht direkt

sieht, weil die Ereignis-Operationen, mit denen sie realisiert werden, in den Spezifikationen der einzelnen Schritte verborgen sind.

Schließlich liegt das Ereigniskonzept auf einer „Bewertungsskala" für Synchronisationsmethoden etwa auf der gleichen Ebene wie einfache Semaphoroperationen [Dijkstra68ab] (d. h. relativ weit „unten"), deren ausschließliche und unmittelbare Verwendung zur Synchronisation nebenläufiger Prozesse bekanntlich sehr schnell zu unübersichtlichen und schwer pflegbaren Formulierungen führt [Freisleben87]. Dies liegt unter anderem darin begründet, daß bei einer solchen Vorgehensweise die Details der Prozeß- bzw. Workflow-*Synchronisation* direkt mit den eigentlichen Prozeß- bzw. Workflow-*Beschreibungen* „verwoben" sind.

1.3.3 Separate Spezifikation von Inter-Workflow-Abhängigkeiten

1.3.3.1 Ansatz

Aus den soeben erläuterten Gründen besteht ein erster wichtiger Schritt zu einer wirklich praktikablen Lösung des Workflow-Synchronisationsproblems darin, die Spezifikation der Inter-Workflow-Abhängigkeiten von der Beschreibung der einzelnen Workflows zu *trennen*. Das bedeutet, daß man das in § 1.1.3 erwähnte *Grundprinzip von Workflow-Management*, nämlich die Trennung der Gesamtablauf-Beschreibung von der Implementierung der einzelnen Schritte, eine Ebene höher erneut anwendet: Man trennt die Spezifikation der Inter-Workflow-Abhängigkeiten von der Modellierung der einzelnen Workflows und erhöht damit sowohl die Übersichtlichkeit als auch die Änderungsfreundlichkeit beider Arten von Beschreibungen.

Abbildung 1.6 verdeutlicht diesen Ansatz für das Sonographie/Endoskopie-Beispiel. Die beiden Workflows stimmen nun wieder exakt mit den ursprünglichen Definitionen aus Abb. 1.1 und 1.2 überein, und die workflowübergreifenden Kontrollflußkanten wurden durch separate Spezifikationen der Art

$$W_1 : S_1 \rightarrow W_2 : S_2 \quad \text{bzw.} \quad W_1 : S_1 \leftrightarrow W_2 : S_2$$

ersetzt, wobei mit W_i $(i = 1, 2)$ eine bestimmte Workflow-Beschreibung und mit S_i ein konkreter Schritt des Workflows W_i bezeichnet wird. Ein einfacher Pfeil $\rightarrow$ steht für eine gewöhnliche Kontrollflußkante, die besagt, daß der Schritt S_2 des Workflows W_2 erst nach Beendigung von Schritt S_1 des Workflows W_1 ausgeführt werden darf. Der Doppelpfeil $\leftrightarrow$ symbolisiert, daß die Schritte S_1 und S_2 der Workflows W_1 und W_2 nach Möglichkeit gleichzeitig bzw. in einem Schritt ausgeführt werden sollen.

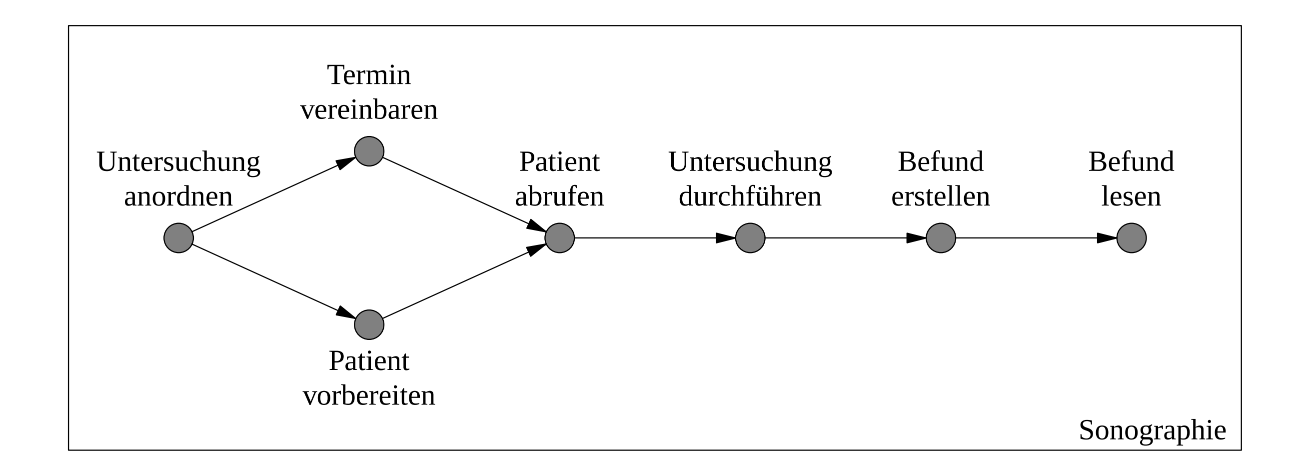
1.3.3.2 Kritik

Obwohl dieser Ansatz den beiden vorangegangenen Lösungsversuchen in punkto Übersichtlichkeit und Änderungsfreundlichkeit eindeutig überlegen ist, wird seine praktische Verwendbarkeit nach wie vor durch eine zu *explizite* und *starre* Formulierung der workflowübergreifenden Synchronisationsbedingungen eingeschränkt. Beispielsweise ist es durch die wechselseitige Abhängigkeit der Workflows voneinander nicht mehr möglich, für einen Patienten *nur* eine Sonographie oder *nur* eine Endoskopie durchzuführen, weil in diesem Fall der entsprechende Workflow vergeblich auf ein Synchronisationsereignis des jeweils anderen Workflows warten würde. Dieses Problem könnte man möglicherweise durch eine flexiblere Interpretation der Inter-Workflow-Abhängigkeiten lösen, indem man Bedingungen, die sich auf einen momentan nicht aktiven Workflow beziehen, außer acht läßt.

Ein weiteres Problem ergibt sich jedoch, wenn zwei Untersuchungen U_1 und U_2 aus medizinischer Sicht in *beliebiger Reihenfolge* durchgeführt werden können, weil in diesem Fall weder die Reihenfolge A:

$$U_1 : \text{Pat. abrufen} \rightarrow U_1 : \text{Unt. durchführen} \rightarrow U_2 : \text{Pat. abrufen} \rightarrow U_2 : \text{Unt. durchführen}$$

noch die umgekehrte Reihenfolge B:



Inter-Workflow-Abhängigkeiten

Endoskopie: Patient vorbereiten	←→	Sonographie: Patient vorbereiten
Endoskopie: Patient vorbereiten	→	Sonographie: Patient abrufen
Sonographie: Untersuchung durchführen	→	Endoskopie: Patient abrufen

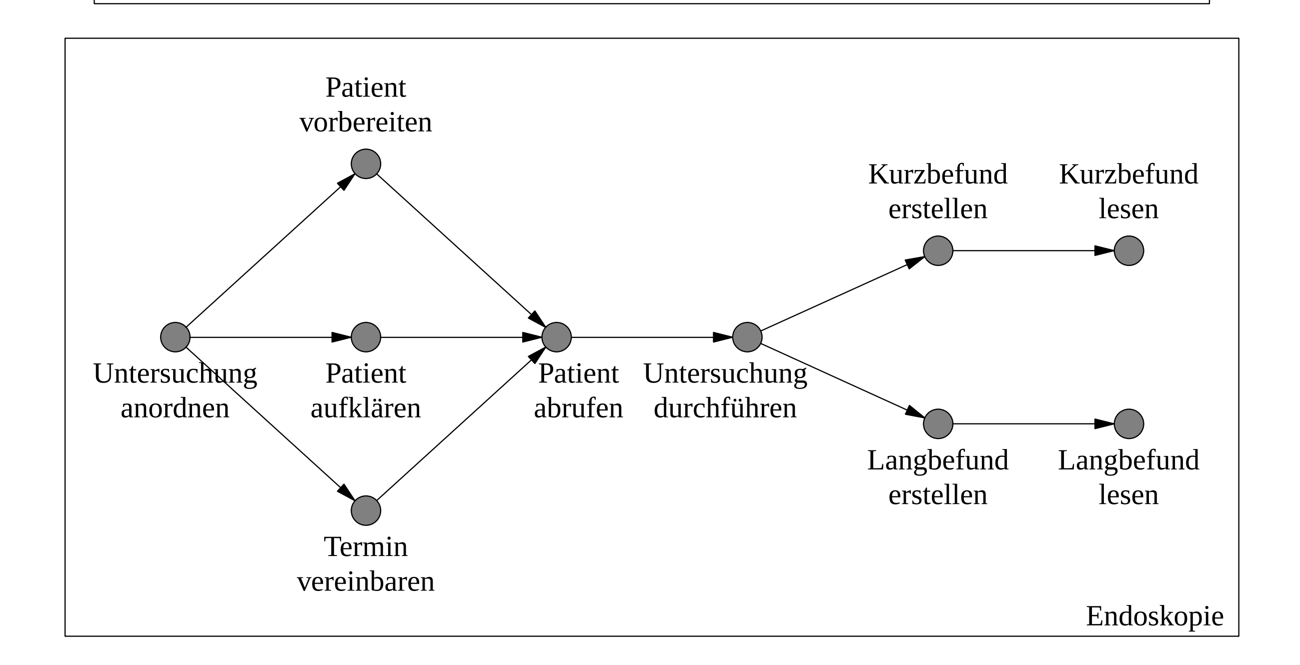


Abbildung 1.6: Separate Spezifikation von Inter-Workflow-Abhängigkeiten

U_2: Pat. abrufen $\rightarrow$ U_2: Unt. durchführen $\rightarrow$ U_1: Pat. abrufen $\rightarrow$ U_1: Unt. durchführen

a priori vorgeschrieben werden kann. Hier bräuchte man einen *Auswahloperator*, etwa $A \mid B$, mit dem spezifiziert werden kann, daß beide Reihenfolgen prinzipiell *gleichberechtigt* sind und daß die Entscheidung für die eine oder andere *implizit* dadurch getroffen wird, welcher der Schritte U_i: Patient abrufen *zuerst* ausgeführt wird. Ein solches Verhalten läßt sich jedoch weder mit den bisher verwendeten Kontrollflußkanten noch mit sonstigen imperativen Konstrukten beschreiben.

Schließlich ist durch eine rein textuelle Trennung der Inter-Workflow-Abhängigkeiten von den einzelnen Workflow-Beschreibungen auch das früher bereits erwähnte *Komplexitätsproblem* noch nicht gelöst, weil auch hier die Menge der zu verwaltenden Bedingungen sehr schnell Größenordnungen an-

nehmen kann, die nicht mehr beherrschbar sind, sobald nicht nur zwei, sondern fünf, zehn oder noch mehr voneinander abhängige Workflows synchronisiert werden müssen. Als „abschreckendes Beispiel" versuche man lediglich, für fünf verschiedene Untersuchungsarten (von denen jede in einem konkreten Ablaufszenario auch fehlen kann) sämtliche zulässigen Ausführungsreihenfolgen der Schritte Patient abrufen und Untersuchung durchführen aufzuzählen.

1.3.4 Deskriptive Spezifikation von Inter-Workflow-Abhängigkeiten

1.3.4.1 Ansatz

Die zuletzt beschriebenen Probleme haben deutlich gemacht, daß die bisher verwendeten *präskriptiven* Methoden [Jablonski95a] zur Spezifikation von Inter-Workflow-Abhängigkeiten ungeeignet sind, weil sie Ausführungsreihenfolgen zu explizit und starr vorschreiben. Die Einführung des Auswahloperators $A \mid B$, der im Prinzip von *regulären Ausdrücken* [Hopcroft90, Schöning95] übernommen wurde, stellt jedoch einen ersten Schritt in Richtung einer stärker *deskriptiv* ausgerichteten Modellierung dar, weil er zwar eine Menge prinzipiell zulässiger Ausführungsreihenfolgen *be*schreibt, jedoch keine bestimmte Reihenfolge *vor*schreibt.

Übernimmt man von regulären Ausdrücken auch noch den *Iterationsoperator* $*$, so läßt sich das eben noch fast unlösbar erscheinende Synchronisationsproblem der Schritte Patient abrufen und Untersuchung durchführen einfach wie folgt lösen: Mit Hilfe des regulären Ausdrucks

$$(U\text{: Patient abrufen} \rightarrow U\text{: Untersuchung durchführen})^*$$

und der im Augenblick noch informellen Vereinbarung, daß die Variable U in jedem Iterationsschritt für eine *beliebige* Untersuchungsart stehen kann, wird auf eine kompakte, übersichtliche und *allgemeingültige* Art und Weise beschrieben, daß die Schritte Patient abrufen und Untersuchung durchführen für einen bestimmten Patienten strikt sequentiell ausgeführt werden müssen.

Auch die weitergehende Bedingung, daß ein Patient z. B. nicht gleichzeitig untersucht und für eine andere Untersuchung vorbereitet werden kann, läßt sich mit Hilfe eines regulären Ausdrucks wie folgt beschreiben:

$$(U\text{: Patient vorbereiten} \mid (U\text{: Patient abrufen} \rightarrow U\text{: Untersuchung durchführen}))^*. \qquad \text{(X)}$$

Allerdings hat man jetzt indirekt auch festgelegt, daß mehrere Ausprägungen des Schritts Patient vorbereiten ebenfalls strikt sequentiell ausgeführt werden müssen, was nach § 1.2 gerade *nicht* beabsichtigt ist. Um auszudrücken, daß der Schritt Patient vorbereiten für einen bestimmten Patienten *beliebig oft gleichzeitig* ausgeführt werden darf, benötigt man einen zusätzlichen Operator (vgl. § 2.3.1.3 und § 3.3.1.7), den es so in regulären Ausdrücken nicht gibt.

1.3.4.2 Kritik

Diese wenigen informellen Beispiele sollen genügen, um zu zeigen, daß ein *deskriptiver Formalismus* im Stil regulärer Ausdrücke zur Beschreibung von Inter-Workflow-Abhängigkeiten *grundsätzlich gut geeignet* ist, daß reguläre Ausdrücke in ihrer reinen Form für praktische Anwendungen jedoch zu *eingeschränkt* sind. Außerdem ist zu berücksichtigen, daß die übliche mathematische Notation regulärer Ausdrücke, wie sie oben verwendet wurde, für entsprechend geschulte Personen zwar sehr elegant und kompakt ist, mathematisch weniger geübten Anwendern (wie z. B. Workflow-Modellierern) aber nicht ohne weiteres zugemutet werden kann. Aus diesem Grund sollten Integritätsbedingungen − ebenso wie Workflowbeschreibungen − in einer möglichst intuitiven *graphischen Notation* formuliert werden können (vgl. z. B. Abb. 1.7), die bei Bedarf in eine äquivalente formale Notation überführt werden kann.

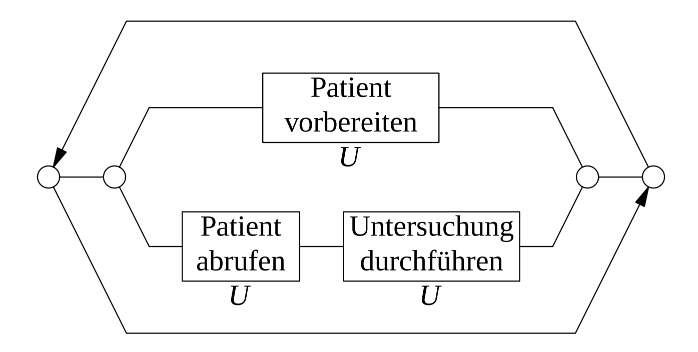


Abbildung 1.7: Graphische Darstellung des regulären Ausdrucks (X)

1.4 Aufgabenstellung und Gliederung der Arbeit

1.4.1 Aufgabenstellung

Vor diesem Hintergrund läßt sich die in § 1.2.3 bereits erwähnte Aufgabenstellung nun wie folgt präzisieren:

- Es soll ein auf regulären Ausdrücken basierender *deskriptiver Formalismus* entwickelt werden, mit dessen Hilfe Inter-Workflow-Abhängigkeiten *kompakt*, *übersichtlich* und *änderungsfreundlich* beschrieben werden können.
- Da der Formalismus auch für mathematisch ungeübte Anwender, wie z. B. Workflow-Modellierer, verständlich sein soll, wird neben einer kompakten mathematischen Notation auch eine äquivalente *graphische Repräsentation* des Formalismus benötigt.
- Ähnlich wie reguläre Ausdrücke, sollte der Formalismus *effizient implementierbar* sein, damit er nicht nur zur formalen Spezifikation, sondern auch zur realen Implementierung von Inter-Workflow-Abhängigkeiten eingesetzt werden kann. Neben einer präzisen *formalen Semantik* soll daher auch eine möglichst effiziente und anhand dieser Semantik verifizierte *Implementierung* des Formalismus erstellt werden.
- Schließlich sollen Mittel und Wege aufgezeigt werden, wie diese Implementierung mit vorhandenen Workflow-Management-Systemen *integriert* werden kann, um sicherzustellen, daß die spezifizierten Inter-Workflow-Abhängigkeiten während der Ausführung von Workflows tatsächlich berücksichtigt werden.

Anmerkung: Der zu entwickelnde Formalismus und seine Implementierung sollen die Build- und Runtime-Komponenten eines WfMSs nicht etwa ersetzen, sondern *ergänzen*, indem sie zusätzliche Integritätsbedingungen spezifizieren und implementieren, die bei der Ausführung von Workflows zu berücksichtigen sind. Einzelne Workflows sollen also nach wie vor mit Hilfe eines Workflow-Editors erstellt und durch eine Workflow-Ausführungseinheit ausgeführt werden. Allerdings kann die Menge der Arbeitsschritte, die aus Sicht eines einzelnen Workflows zu einem bestimmten Zeitpunkt zulässig sind, durch Workflow-übergreifende Integritätsbedingungen wie in Abb. 1.7 mehr oder weniger stark *eingeschränkt* werden. Die zu entwickelnde Implementierung stellt somit einen zusätzlichen *Interaktions-Manager* dar, der entweder in eine Workflow-Ausführungseinheit integriert ist oder in geeigneter Weise mit dieser kommuniziert (vgl. § 5.5.3).

Außerdem wird in § 5.3 gezeigt, daß der Formalismus nicht nur zur Spezifikation und Implementierung von Inter-Workflow-Abhängigkeiten, sondern z. B. auch zur Synchronisation paralleler Programme eingesetzt werden kann.

1.4.2 Gliederung

Entsprechend dieser Aufgabenstellung befaßt sich der Hauptteil der Arbeit mit den verschiedensten Aspekten des genannten Formalismus, die nacheinander wie folgt behandelt werden:

- In Kapitel 2 werden *Interaktionsgraphen* als *graphischer Formalismus* zur Beschreibung von Abhängigkeiten oder Wechselwirkungen zwischen Tätigkeiten (lat. *inter actiones*) eingeführt. Anhand zahlreicher Beispiele wird schrittweise erläutert, wie Interaktionsgraphen *aufgebaut* sind und nach welchen Regeln sie *traversiert* bzw. *durchlaufen* werden. Gegen Ende des Kapitels (§ 2.7) wird auch das zuvor skizzierte Anwendungsszenario der interagierenden medizinischen Untersuchungsworkflows wiederaufgegriffen und anhand von Beispielgraphen illustriert, wie die benötigten Integritätsbedingungen mit Hilfe von Interaktionsgraphen spezifiziert werden können.

- In Kapitel 3 werden *Interaktionsausdrücke* als *mathematischer Formalismus* mit einer exakt definierten Semantik eingeführt, der konzeptionell äquivalent zu Interaktionsgraphen ist. Auf diese Weise wird indirekt auch die Semantik von Interaktionsgraphen präzise definiert, die in Kapitel 2 bewußt nur informell und anschaulich festgelegt wurde. Außerdem werden in diesem Kapitel zahlreiche *formale Eigenschaften* sowie Aussagen zur *Ausdrucksmächtigkeit* von Interaktionsausdrücken vorgestellt.

- Nach diesen grundlegenden Kapiteln wird in Kapitel 4 eine konkrete *Implementierung* von Interaktionsausdrücken entwickelt, die in der Lage ist, für einen gegebenen Ausdruck das klassische *Wortproblem* [Hopcroft90, Schöning95], das verwandte *Teilwortproblem* sowie das für konkrete Anwendungen essentielle *Aktionsproblem* (vgl. § 4.1.1) zu lösen. Da sich die in Kapitel 3 vorgestellte formale Semantik nicht direkt in eine ausreichend effiziente Implementierung transformieren läßt, besteht ein wesentlicher Teil dieses Kapitels (§ 4.5) in der Entwicklung einer separaten *operationalen Semantik*, die einerseits äquivalent zur formalen Semantik ist und andererseits *effizient* implementierungstechnisch umgesetzt werden kann. Das Kapitel schließt mit einigen *Komplexitätsbetrachtungen*, die formal bestätigen, daß die so entwickelte Implementierung eine sehr große und praktisch relevante Teilklasse von Interaktionsausdrücken effizient verarbeiten kann.

- Im Anschluß an diese ausführliche, sowohl theoretisch als auch praktisch relevante Aspekte umfassende Behandlung von Interaktionsausdrücken und -graphen, wird in Kapitel 5 die Brücke zurück zur *Anwendung* geschlagen. Insbesondere wird erläutert, wie Interaktionsgraphen zur Spezifikation von *Inter-Workflow-Abhängigkeiten* eingesetzt werden können und wie die in Kapitel 4 vorgestellte Implementierung zu einem *Interaktions-Manager* ausgebaut und mit einem oder mehreren Workflow-Management-Systemen gekoppelt werden kann, um die Einhaltung der spezifizierten Bedingungen während der Ausführung von Workflows sicherzustellen.

- In Kapitel 6 werden zahlreiche *verwandte Arbeiten* vorgestellt und mit Interaktionsausdrücken verglichen. Hierbei wird deutlich, daß die grundsätzliche Idee, erweiterte reguläre Ausdrücke zur Beschreibung von Synchronisationsbedingungen zu verwenden, zwar nicht neu ist, daß die bisher in der Literatur vorgeschlagenen Ansätze jedoch alle mehr oder weniger lückenhaft sind und Interaktionsausdrücke somit eine *Vereinigung* und *Erweiterung* vieler anderer Formalismen darstellen. Außerdem wird erläutert, wie sich Interaktionsausdrücke im Detail von potentiell *nichtdeterministischen* Formalismen wie Prozeßalgebren und Petrinetzen unterscheiden.

- In Kapitel 7 schließlich werden die wesentlichen Ergebnisse und Errungenschaften der Arbeit zusammengefaßt und ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

Abbildung 1.8 stellt die Struktur der Arbeit graphisch dar und deutet an, daß sich ihr Inhalt grob in drei Abstraktionsebenen unterteilen läßt:

1. Sofern man lediglich an den wesentlichen Ideen und Ergebnissen interessiert ist, genügt es, die beiden Kapitel der obersten Ebene zu lesen.

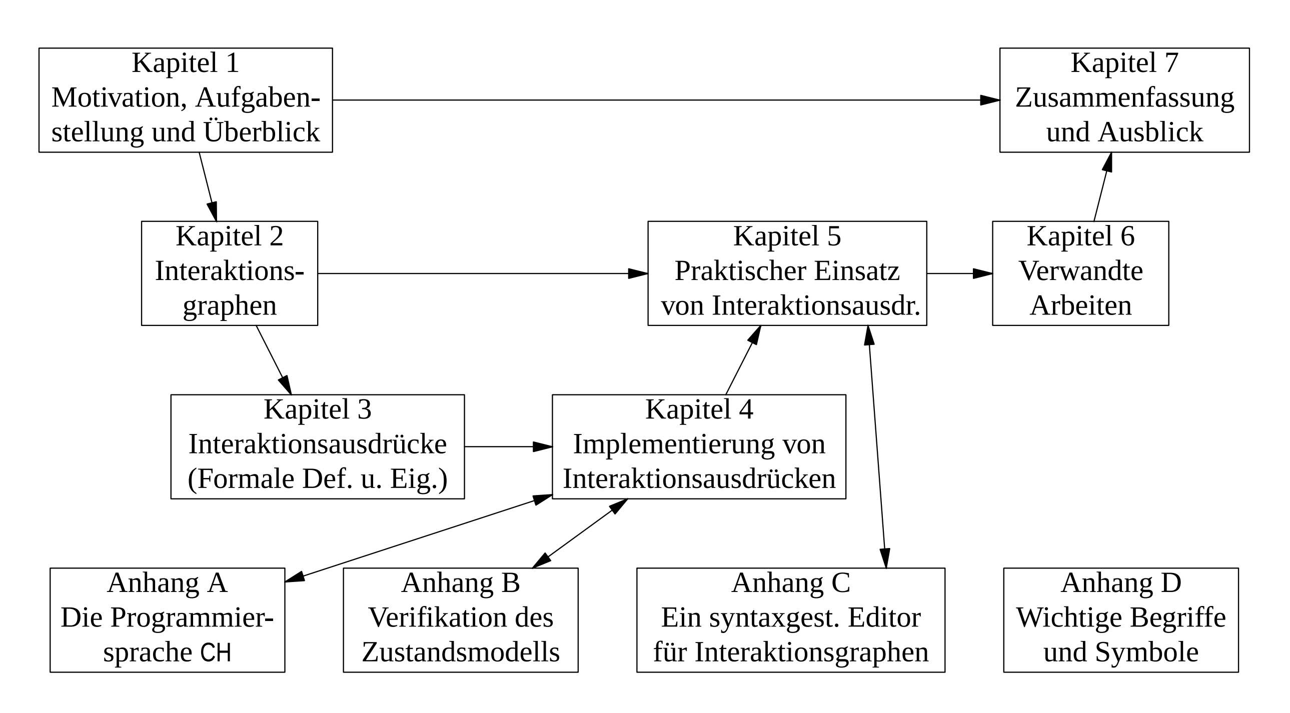


Abbildung 1.8: Gliederung der Arbeit

2. Um Interaktionsgraphen und ihre Einsatzmöglichkeiten praxisorientiert kennenzulernen und ein Gefühl für ihre Möglichkeiten und Grenzen zu entwickeln, sollte man die obersten beiden Ebenen durchlaufen. Bei diesem Rundgang durch die Arbeit, bei dem man den „formalen Unterbau“ einschließlich der Implementierung von Interaktionsausdrücken einfach als gegeben annimmt, bleiben dem Leser formale Details und „Strapazen“ größtenteils erspart, ohne daß dadurch das Gesamtverständnis wesentlich beeinträchtigt wird.

3. Ist man darüber hinaus an formalen oder implementierungstechnischen Details interessiert oder will man sich explizit von deren Korrektheit überzeugen, so sollte man bis auf die dritte Ebene hinabsteigen, d. h. die Arbeit vollständig lesen.

Die in der Graphik erwähnten Anhänge A und B gehören logisch zu Kapitel 4, wurden aber aufgrund ihres Umfangs extrahiert, um den inhaltlichen Fluß dieses Kapitels nicht unnötig zu behindern. Anhang A erläutert kurz die wesentlichen Konzepte der zur Implementierung verwendeten Programmiersprache CH (bei der es sich um eine Variante von C++ handelt), während Anhang B umfangreiche Beweise enthält, die im wesentlichen die Korrektheit der operationalen Semantik (d. h. ihre Äquivalenz zur formalen Semantik) zeigen.

Anhang C beschreibt einen *syntaxgesteuerten Editor* für Interaktionsgraphen und gehört inhaltlich zu Kapitel 5. In Anhang D schließlich werden alle wichtigen Begriffe und Symbole, die in dieser Arbeit eingeführt und verwendet werden, tabellarisch zusammengefaßt.

Kapitel 2

Interaktionsgraphen

2.1 Einleitung

2.1.1 Zielsetzung

Im vorliegenden Kapitel werden *Interaktionsgraphen* als graphischer Formalismus zur Beschreibung von *Abhängigkeiten* oder Wechselwirkungen *zwischen Tätigkeiten* (lat. *inter actiones*) vorgestellt. Im Gegensatz zum nachfolgenden Kapitel 3, in dem die exakte *Semantik* von Interaktionsgraphen (bzw. -ausdrücken) mit Hilfe *formaler Sprachen* spezifiziert wird, werden Interaktionsgraphen in diesem Kapitel *informell* und *anschaulich* eingeführt und erläutert. Um die Bedeutung eines Graphen zu verstehen, stelle man sich vor, daß man ihn − ähnlich wie ein Syntaxdiagramm − nach bestimmten, in den folgenden Abschnitten erläuterten Regeln *von links nach rechts durchläuft* und die Folge der hierbei *passierten Aktionen* als *zulässige Ausführungsreihenfolge* interpretiert. Da die verschiedenen *Verzweigungs-* und *Vereinigungsknoten* von Interaktionsgraphen gewisse *Freiheitsgrade* für das Durchlaufen von Teilgraphen vorsehen, gibt es meist nicht nur einen, sondern *mehrere unterschiedliche Wege* durch einen Graphen. Dementsprechend definiert ein Interaktionsgraph im allgemeinen eine ganze *Menge* zulässiger Ausführungsreihenfolgen, und in vielen Fällen ist diese Menge sogar *unendlich* groß.

2.1.2 Überblick

In den grundlegenden Abschnitten 2.2 und 2.3 wird anhand zahlreicher Beispiele − die zunächst noch nichts mit der konkreten Anwendungsdomäne Inter-Workflow-Abhängigkeiten zu tun haben − schrittweise erläutert, wie Interaktionsgraphen *aufgebaut* sind und nach welchen Regeln sie *traversiert* bzw. *durchlaufen* werden.

Anschließend werden in den Abschnitten 2.4 bis 2.6 einige wichtige Detailfragen erörtert: Zunächst wird ausführlich erläutert, wie Graphen *zielgerichtet* durchlaufen werden, welche Schwierigkeiten das hierbei zu berücksichtigende Prinzip des *deterministischen Verhaltens* verursachen kann und wie diese prinzipiell überwunden werden (§ 2.4). Anschließend wird das Phänomen von *Sackgassen* und *endlosen Wegen* diskutiert (§ 2.5) und schließlich der Unterschied zwischen punktuellen *Aktionen* und zeitlich ausgedehnten *Aktivitäten* erläutert (§ 2.6).

Das in Kapitel 1 vorgestellte Anwendungsszenario der interagierenden medizinischen Untersuchungsworkflows wird erst in § 2.7 wieder aufgegriffen, nachdem in den vorangegangenen Abschnitten alle wesentlichen Konzepte von Interaktionsgraphen ausführlich besprochen wurden.

In § 2.8 schließlich werden sowohl die *Operatoren* als auch die wesentlichen Prinzipien von Interaktionsgraphen noch einmal zusammengefaßt.

2.2 Grundlegende Operatoren (Beispiel Münzautomaten)

2.2.1 Einwerfen von 2 DM

Die Interaktionsgraphen in Abb. 2.1 und 2.2 zeigen zwei verschiedene Varianten, einen Betrag von 2 DM in einen Münzautomaten einzuwerfen, der 1-DM- und 2-DM-Münzen akzeptiert, d. h. der grundsätzlich die Ausführung der Aktionen 1 DM (Einwerfen einer 1-DM-Münze) und 2 DM (Einwerfen einer 2-DM-Münze) erlaubt: Entweder man wirft eine einzelne 2-DM-Münze ein (Abb. 2.1) oder aber zwei 1-DM-Münzen nacheinander (Abb. 2.2). Um mit einem einzigen Graphen auszudrücken,

2 DM

Abbildung 2.1: Einwerfen von 2 DM (Variante 1)

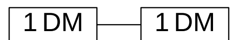


Abbildung 2.2: Einwerfen von 2 DM (Variante 2)

daß man *entweder* einmal die Aktion 2 DM *oder* zweimal nacheinander die Aktion 1 DM ausführen kann, werden die beiden Graphen mittels einer *Entweder-oder-Verzweigung* wie in Abb. 2.3 verknüpft.

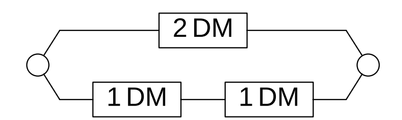


Abbildung 2.3: Einwerfen von 2 DM (Kombination von Variante 1 und 2)

Eine solche Verzweigung wird durchlaufen, indem man am Verzweigungsknoten ○ (links) *entweder* nach oben *oder* nach unten abbiegt und somit genau *einen* der beiden Teilgraphen bzw. *Zweige* zwischen den beiden ○-Knoten durchläuft. Je nachdem, für welchen Zweig man sich entscheidet, durchläuft man entweder einmal die Aktion 2 DM (oben) oder aber zweimal hintereinander die Aktion 1 DM (unten), bevor man den Vereinigungsknoten ○ (rechts) und somit das Ende des Graphen erreicht.

2.2.2 Einwerfen von 3 DM

2.2.2.1 Variante 1

Um einen Betrag von 3 DM mit Hilfe von 1-DM- und 2-DM-Münzen zu entrichten, kann man entweder drei 1-DM-Münzen oder aber eine 1-DM- und eine 2-DM-Münze in beliebiger Reihenfolge einwerfen. Dies wird durch den Graphen in Abb. 2.4 beschrieben.

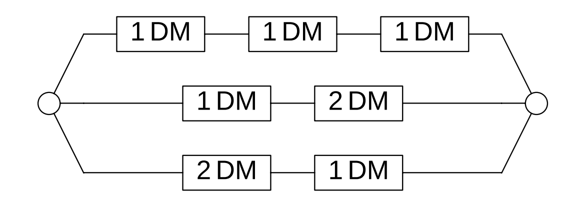


Abbildung 2.4: Einwerfen von 3 DM (Variante 1)

Eine Entweder-oder-Verzweigung kann also nicht nur zwei, sondern prinzipiell *beliebig viele Zweige* besitzen, von denen genau *einer* zu durchlaufen ist.

2.2.2.2 Variante 2

Offensichtlich äquivalent zu dieser Darstellung ist der Graph in Abb. 2.5, bei dem die ternäre Verzweigung aus Abb. 2.4 durch eine Verschachtelung zweier binärer Verzweigungen ersetzt wurde: Am ersten Verzweigungsknoten (ganz links) kann man entweder nach oben oder nach unten abbiegen; entscheidet man sich für unten, so kann man am nächsten (inneren) Verzweigungsknoten erneut zwischen oben und unten wählen. Folglich durchläuft man auch hier genau eine der drei Sequenzen 1 DM – 1 DM – 1 DM, 1 DM – 2 DM oder 2 DM – 1 DM.

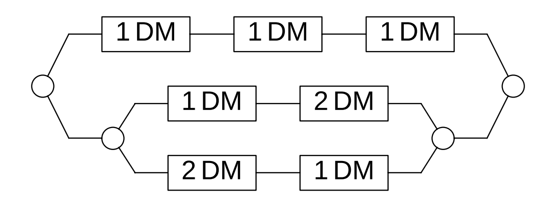


Abbildung 2.5: Einwerfen von 3 DM (Variante 2)

2.2.2.3 Variante 3

Eine dritte äquivalente Darstellung erhält man, wenn man die innere Entweder-oder-Verzweigung in Abb. 2.5 durch eine *Sowohl-als-auch-Verzweigung* wie in Abb. 2.6 ersetzt. Eine solche Verzweigung − deren Knoten im Gegensatz zur Entweder-oder-Verzweigung durch Doppelkreise ◎ dargestellt werden − wird durchlaufen, indem man am Verzweigungsknoten ◎ (links) *sowohl* nach oben *als auch* nach unten abbiegt und *beide* Zweige parallel, d. h. *unabhängig voneinander* durchläuft. Je nachdem, ob hierbei zuerst im oberen Zweig die Aktion 1 DM oder im unteren Zweig die Aktion 2 DM durchlaufen wird, erhält man für den Teilgraphen ◎···◎ eine der Ausführungsreihenfolgen 1 DM – 2 DM oder 2 DM – 1 DM. Da man in einen Automaten normalerweise nicht mehrere Münzen auf einmal einwerfen kann (weil er nur einen Münzschlitz besitzt), muß man sich für eine dieser beiden Reihenfolgen entscheiden, d. h. man kann die Aktionen 1 DM und 2 DM *nicht gleichzeitig* ausführen.[1]

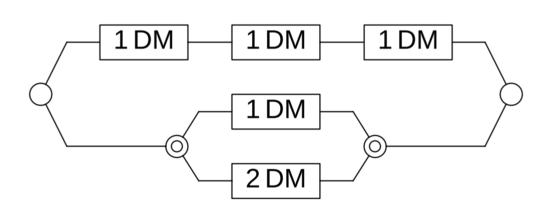


Abbildung 2.6: Einwerfen von 3 DM (Variante 3)

Anmerkung zur graphischen Darstellung: Ein *Doppel*-Kreis ◎ (Sowohl-als-auch-Verzweigung) symbolisiert, daß *beide* (bzw. alle) Zweige der Verzweigung zu durchlaufen sind, während ein *einfacher* Kreis ○ (Entweder-oder-Verzweigung) anzeigt, daß genau *ein* Zweig gewählt werden muß.

[1] In § 2.6.1 wird dieses Prinzip auf beliebige Aktionen verallgemeinert, d. h. es wird generell vereinbart, daß zwei Aktionen niemals gleichzeitig ausgeführt werden können.

2.2.2.4 Sowohl-als-auch-Verzweigungen

Zum besseren Verständnis der Sowohl-als-auch-Verzweigung betrachte man den Graphen in Abb. 2.7, in dem die Buchstaben a bis e fünf nicht näher spezifizierte Aktionen repräsentieren. Um den Graphen zu durchlaufen, biegt man am linken ◎-Knoten, wie bereits erläutert, sowohl nach oben als auch nach unten ab und durchläuft die Teilgraphen $a - b$ und $c - d$ unabhängig voneinander. Abhängig von den „relativen Geschwindigkeiten“, mit denen die beiden Zweige durchlaufen werden, können die Aktionen a bis e in genau einer der folgenden Reihenfolgen passiert werden:

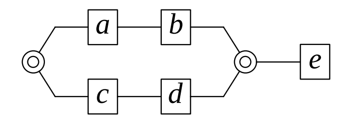


Abbildung 2.7: Sowohl-als-auch-Verzweigung

1. Im oberen Zweig wird a passiert, bevor im unteren Zweig c passiert wird (a vor c).
   a) Im oberen Zweig wird b passiert, bevor im unteren Zweig c passiert wird (b vor c).
   Da der obere Zweig jetzt vollständig durchlaufen ist, muß man am Vereinigungsknoten ◎ (rechts) warten, bis auch der untere Zweig vollständig durchlaufen ist, d. h. jetzt müssen die Aktionen c und d des unteren Zweigs durchlaufen werden. Erst dann kann die der Sowohl-als-auch-Verzweigung folgende Aktion e passiert werden.
   Die resultierende Gesamtreihenfolge ist daher $a - b - c - d - e$.
   b) Im unteren Zweig wird c passiert, bevor im oberen Zweig b passiert wird (c vor b).
      α) Im oberen Zweig wird b passiert, bevor im unteren Zweig d passiert wird (b vor d).
      Da der obere Zweig jetzt vollständig durchlaufen ist, muß jetzt im unteren Zweig d passiert werden, bevor die Folgeaktion e passiert werden kann.
      Resultat: $a - c - b - d - e$.
      β) Im unteren Zweig wird d passiert, bevor im oberen Zweig b passiert wird (d vor b).
      Symmetrisch zum letzten Fall, muß jetzt im oberen Zweig b passiert werden, bevor die Folgeaktion e passiert werden kann.
      Resultat: $a - c - d - b - e$.

2. Im unteren Zweig wird c passiert, bevor im oberen Zweig a passiert wird (c vor a).
   Symmetrisch zum Fall 1, sind jetzt die folgenden Varianten möglich:
   a) d vor a.
   Resultat: $c - d - a - b - e$.
   b) a vor d.
      α) d vor b.
      Resultat: $c - a - d - b - e$.
      β) b vor d.
      Resultat: $c - a - b - d - e$.

Die Menge der zulässigen Ausführungsreihenfolgen erhält man also, indem man die Folgen $a - b$ und $c - d$ beliebig miteinander *verschränkt* und anschließend jeweils ein e anfügt. Das Verschränken zweier Folgen ist vergleichbar mit dem Reißverschlußverfahren an einer Autobahn-Engstelle oder dem Zusammenmischen zweier Kartenstapel, bei dem in jedem Verarbeitungsschritt *zufällig* das nächste Element *einer* der beiden Sequenzen (Aktionsfolgen, Autoschlangen oder Kartenstapel) zur resultierenden Sequenz hinzugefügt wird (vgl. Abb. 2.8). Da das Schema, nach dem die beiden gege-

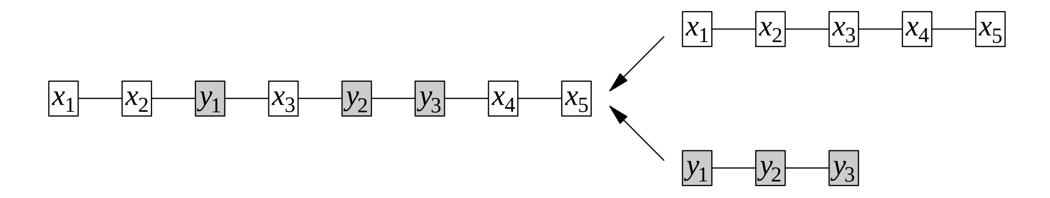


Abbildung 2.8: Beispiel einer Verschränkung

benen Sequenzen abgearbeitet werden, nicht exakt vorgegeben ist, sind prinzipiell verschiedene „Vermischungen“ möglich. Da beide Sequenzen jedoch von links nach rechts abgearbeitet werden, bleibt ihre *relative Ordnung* in jedem Fall erhalten, d. h. ein Element a, das in einer der gegebenen Sequenzen vor einem anderen Element a' steht, wird auch in jeder resultierenden Sequenz vor a' stehen.

2.2.2.5 Variante 4

Eine vierte Lösung des 3-DM-Problems basiert im Prinzip auf der Gleichung 3 = 2 + 1, d. h. auf der Beobachtung, daß man *sowohl* 2 DM *als auch* 1 DM einwerfen muß, um einen Betrag von 3 DM zu entrichten. Da die Teilprobleme „2 DM einwerfen“ (vgl. Abb. 2.3) und „1 DM einwerfen“ (trivial) bereits gelöst sind und die zugehörigen Aktionen offensichtlich *unabhängig voneinander* ausgeführt werden können, kann man die Graphen der beiden Teilprobleme einfach mittels einer Sowohl-als-auch-Verzweigung verknüpfen, um eine Lösung für das Gesamtproblem „3 DM einwerfen“ zu erhalten (vgl. Abb. 2.9).

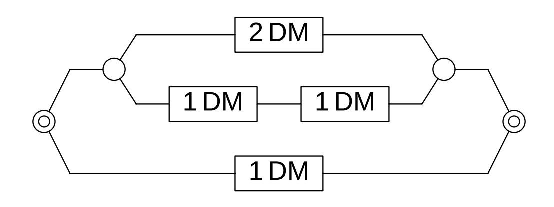


Abbildung 2.9: Einwerfen von 3 DM (Variante 4)

Um diesen Graphen zu durchlaufen, biegt man am linken ◎-Knoten *sowohl* nach oben *als auch* nach unten ab. Im oberen Zweig muß man dann am linken ○-Knoten *entweder* nach oben *oder* nach unten abbiegen. Biegt man nach oben ab, so erhält man insgesamt eine der beiden Reihenfolgen 1 DM – 2 DM oder 2 DM – 1 DM, je nachdem ob man zuerst die Aktion 1 DM des unteren ◎-Zweigs oder die Aktion 2 DM durchläuft. Biegt man an der ○-Verzweigung nach unten ab, so erhält man insgesamt die Reihenfolge 1 DM – 1 DM – 1 DM, und zwar unabhängig davon, ob die Aktion 1 DM des unteren ◎-Zweigs im ersten, zweiten oder dritten Schritt durchlaufen wird.

2.2.3 Einwerfen von 4 DM

2.2.3.1 Variante 1

Der zuletzt vorgestellte Lösungsansatz, bei dem zwei unabhängig voneinander entwickelte Teilgraphen zu einem größeren Graphen zusammengefügt wurden, läßt sich unmittelbar auf das Problem „4 DM einwerfen“ übertragen: Aufgrund der Gleichung 4 = 2 + 2 genügt es, zwei Ausprägungen des Graphen aus Abb. 2.3 (Einwerfen von 2 DM) mittels einer Sowohl-als-auch-Verzweigung zu verknüpfen (vgl. Abb. 2.10).

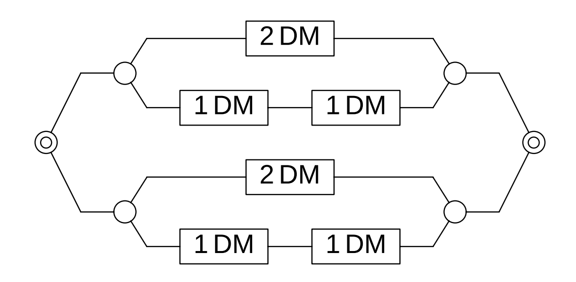


Abbildung 2.10: Einwerfen von 4 DM (Variante 1)

Durch Ausprobieren sämtlicher möglicher Durchlaufreihenfolgen kann man sich in der Tat davon überzeugen, daß dieser Graph alle denkbaren Möglichkeiten beschreibt, einen Betrag von 4 DM mit Hilfe von 1-DM- und 2-DM-Münzen zu entrichten.

2.2.3.2 Variante 2

Da die beiden Zweige der Sowohl-als-auch-Verzweigung in Abb. 2.10 identisch sind, wäre es übersichtlicher und bequemer, wenn man diesen Teilgraphen nur *einmal* formulieren müßte. Mit Hilfe einer *Mehrfach-Verzweigung* wie in Abb. 2.11 ist dies möglich: Der *Verzweigungsfaktor* 2 oberhalb der beiden großen ◎-Knoten besagt, daß der Teilgraph zwischen diesen beiden Knoten „in Wirklichkeit" zweimal vorhanden ist.

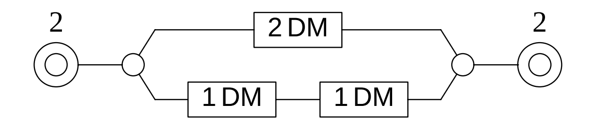


Abbildung 2.11: Einwerfen von 4 DM (Variante 2)

2.2.3.3 Mehrfach-Verzweigungen

Ganz allgemein wird eine Mehrfach-Verzweigung „n-mal x" (vgl. Abb. 2.12) mit einem Verzweigungsfaktor $n \in \mathbb{N}$ und einem beliebigen Teilgraphen x wie folgt durchlaufen:[2]

- Anstelle eines einzelnen *Läufers*, der den Graphen durchläuft, stelle man sich eine *Mannschaft* von Läufern vor, die sich am linken ◎-Knoten in *n Gruppen* aufteilt.[3]
- Jede Gruppe durchläuft den Teilgraphen x unabhängig von den übrigen Gruppen.

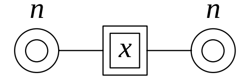


Abbildung 2.12: Mehrfach-Verzweigung „n-mal x"

[2] Da der Bezeichner x hier einen beliebig komplexen Teilgraphen repräsentiert, wird er − im Gegensatz zu einer einfachen Aktion − mit einem Doppelrahmen umgeben (vgl. auch § 2.2.4.4).

[3] Auf den ersten Blick würde es genügen, die Mannschaft in einzelne *Läufer* aufzuteilen. Berücksichtigt man jedoch, daß Mehrfach-Verzweigungen verschachtelt werden können, so benötigt man tatsächlich *Gruppen* von Läufern, die sich an einer inneren Verzweigung erneut in Teilgruppen aufteilen können usw.

- Am rechten ◎-Knoten treffen die Gruppen wieder zusammen und warten aufeinander, d. h. die Gesamtmannschaft setzt ihre „Reise“ erst dann fort, wenn alle Gruppen diesen Synchronisationsknoten erreicht haben.

2.2.4 Einwerfen von 5 DM

2.2.4.1 Variante 1

Eine Lösung für das Problem „Einwerfen von 5 DM“ erhält man nun offensichtlich, indem man die 4-DM-Lösung (Abb. 2.11) − wiederum mit Hilfe einer Sowohl-als-auch-Verzweigung − um einen weiteren 1-DM-Zweig erweitert. Will man darüber hinaus das Einwerfen einer einzelnen 5-DM-Münze akzeptieren, so kann dies durch eine zusätzliche Entweder-oder-Verzweigung dargestellt werden (Abb. 2.13).

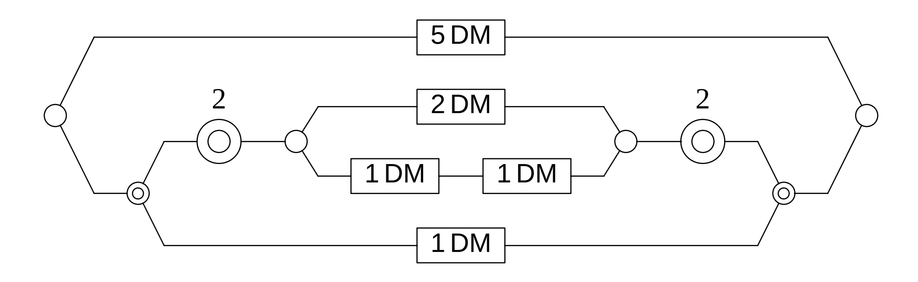


Abbildung 2.13: Einwerfen von 5 DM (Variante 1)

2.2.4.2 Variante 2

Offensichtlich äquivalent zu dieser Darstellung ist der Graph in Abb. 2.14, den man durch folgende Umformungen aus Abb. 2.13 erhält:

1. Die Mehrfach-Verzweigung wird gemäß ihrer Definition zu einer Sowohl-als-auch-Verzweigung expandiert.
2. Die resultierende Verschachtelung zweier binärer Sowohl-als-auch-Verzweigungen wird durch eine äquivalente ternäre Verzweigung ersetzt (was aufgrund des *Assoziativgesetzes* zulässig ist, vgl. § 3.4.7.2).

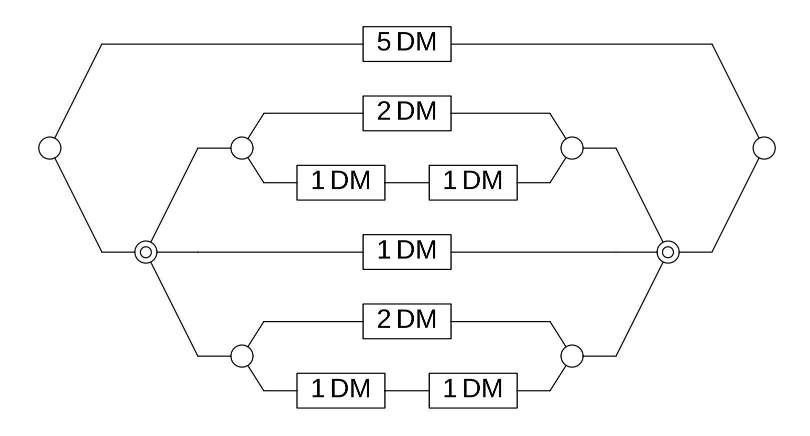


Abbildung 2.14: Einwerfen von 5 DM (Variante 2)

3. Die Zweige dieser ternären Verzweigung werden nach Belieben permutiert (was aufgrund des *Kommutativgesetzes* möglich ist, vgl. ebenfalls § 3.4.7.2).

Ebenso wie eine Entweder-oder-Verzweigung, kann also auch eine Sowohl-als-auch-Verzweigung prinzipiell beliebig viele Zweige besitzen, die alle unabhängig voneinander durchlaufen werden.

2.2.4.3 Kleinere Münzen

Abbildung 2.15 zeigt, daß das 5-DM-Problem auch durch eine explizite Aufzählung aller möglichen Münzfolgen noch zufriedenstellend gelöst werden könnte. Dies ändert sich jedoch schlagartig, wenn der Automat neben 5-DM-, 2-DM- und 1-DM-Münzen auch noch 50-Pfennig- und evtl. sogar 10-Pfennig-Münzen akzeptieren soll, weil dadurch die Anzahl der möglichen Reihenfolgen von neun auf 170 bzw. knapp 2 Millionen ansteigen würde![4] Ausgehend von Abb. 2.13, läßt sich jedoch schrittweise eine wesentlich kompaktere Lösung entwickeln:

1. Zunächst wird die Sequenz 1 DM – 1 DM durch die offensichtlich äquivalente Mehrfach-Verzweigung „2-mal 1 DM“ ersetzt (Abb. 2.16).

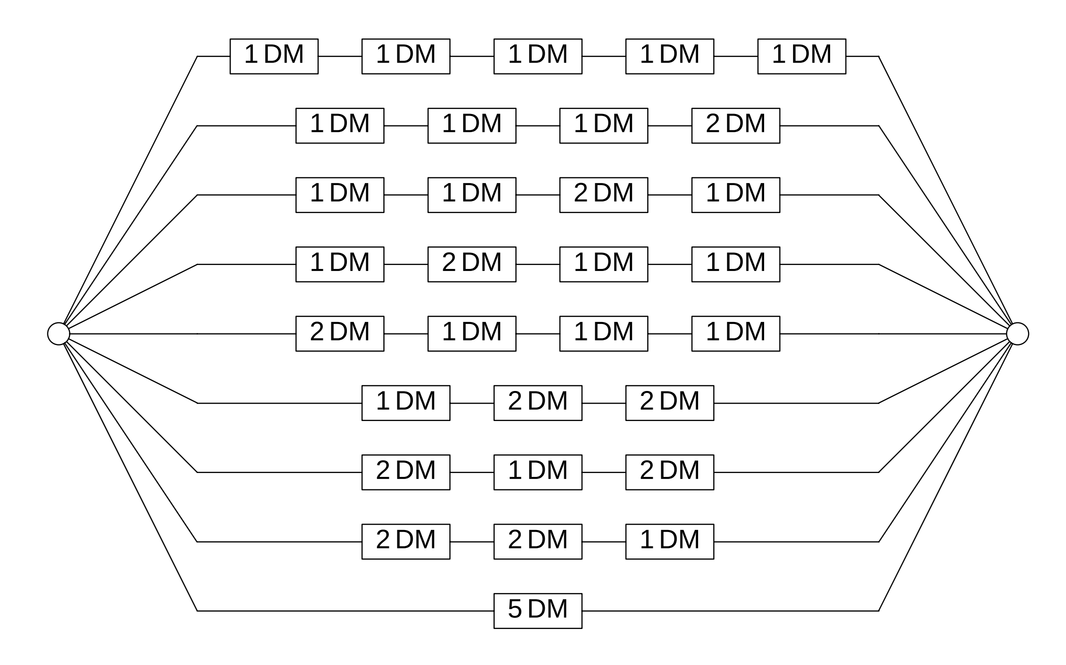


Abbildung 2.15: Einwerfen von 5 DM (Variante 3)

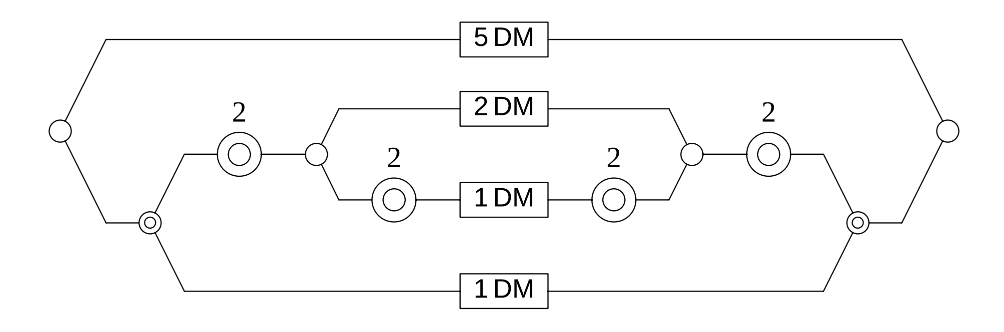


Abbildung 2.16: Einwerfen von 5 DM (Variante 4)

[4] Diese Zahlen lassen sich relativ einfach mit Hilfe eines Programms ermitteln, das systematisch alle möglichen Münzfolgen durchprobiert.

2. Anschließend wird jede 1-DM-Münze durch eine Entweder-oder-Verzweigung „1 DM oder 2-mal 50 Pf“ ersetzt (Abb. 2.17).
3. Schließlich wird jede 50-Pfennig-Münze durch eine Entweder-oder-Verzweigung „50 Pf oder 5-mal 10 Pf“ ersetzt (Abb. 2.18).

Da es, wie oben erwähnt, knapp 2 Millionen Möglichkeiten gibt, einen Betrag von 5 DM mit Hilfe der Münzen 10 Pf, 50 Pf, 1 DM, 2 DM und 5 DM zu entrichten, ist es natürlich nicht mehr möglich, den Graphen in Abb. 2.18 dadurch zu verifizieren, daß man sämtliche möglichen Durchlaufreihenfolgen ausprobiert. Durch die beschriebene „Top-down-Entwicklung“ des Graphen, bei der jeweils ein bestimmter Teilgraph (im Beispiel immer eine einzelne Aktion) durch einen semantisch äquivalenten Teilgraphen ersetzt (bzw. verfeinert) wurde, ist dies jedoch auch nicht erforderlich. Es genügt, daß man von der Korrektheit des Ausgangsgraphen (Abb. 2.13) und von der Korrektheit der einzelnen Transformationsschritte überzeugt ist.

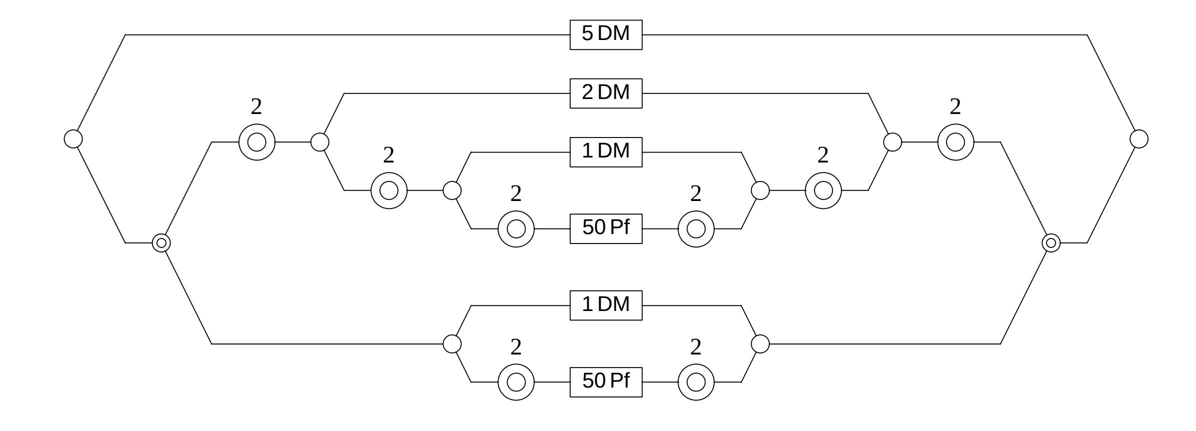


Abbildung 2.17: Einwerfen von 5 DM (Variante 5)

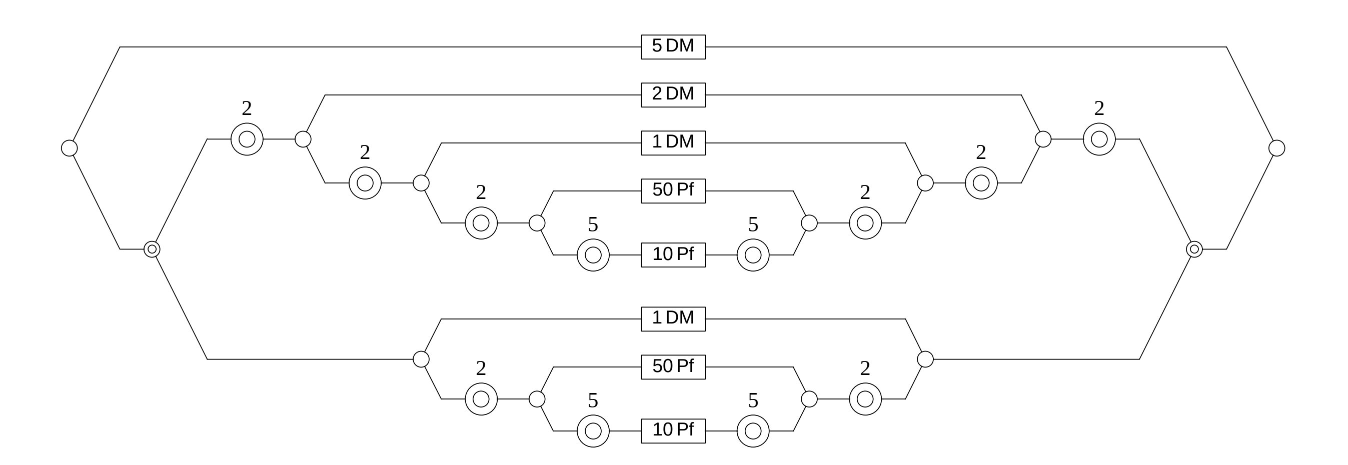


Abbildung 2.18: Einwerfen von 5 DM (Variante 6)

2.2.4.4 Abkürzungen

Ähnlich wie der Graph in Abb. 2.10, enthält auch der soeben entwickelte Graph (Abb. 2.18) einige identische Teilgraphen, die sich jedoch − im Gegensatz zu dort − nicht durch die Verwendung von Mehrfach-Verzweigungen eliminieren lassen. Außerdem leidet seine Übersichtlichkeit und Verständlichkeit an der relativ großen Verschachtelungstiefe der verschiedenen Verzweigungen, die darüber hinaus die Verwendung einer sehr kleinen Schrift erzwingt, damit der Graph horizontal auf eine Seite paßt.

Beide genannten „Schönheitsfehler" lassen sich durch die Einführung von *Abkürzungen* vermeiden: Zum einen kann man Teilgraphen, die mehrfach benötigt werden, an Bezeichner zuweisen, die anschließend als Stellvertreter dieser Teilgraphen dienen und wie einfache Aktionen verwendet werden können; auf diese Weise erspart man sich das unbequeme und fehleranfällige Replizieren von Teilgraphen. Zum anderen ist es möglich, die Lösung eines Problems schrittweise (entweder „top down" oder „bottom up") zu entwickeln und so eine übermäßige Verschachtelung von Verzweigungen zu vermeiden.

Abbildung 2.19 zeigt diese Vorgehensweise für die Lösung des 5-DM-Problems. Abkürzungen werden, ähnlich wie Aktionen, durch Rechtecke dargestellt, der Doppelrahmen deutet jedoch an, daß es sich in Wirklichkeit um beliebig komplexe Teilgraphen handelt. „Liest" man die Abbildung „zeilenweise" von oben nach unten und ersetzt dabei gedanklich jede Abkürzung durch ihre zuvor formulierte Definition, so erhält man in der letzten „Zeile" (Abkürzung Fünf Mark) genau denselben Graphen wie in Abb. 2.18.

Anmerkung: Da Interaktionsgraphen grundsätzlich *beliebig* kombiniert und verschachtelt werden dürfen (vgl. auch § 2.8.2.1), gibt es keine Einschränkungen, was die Struktur oder Komplexität von Abkürzungen betrifft, solange sie sich durch einfachen „Textersatz" − *ohne rekursive oder zyklische Abhängigkeiten* − auflösen lassen.[5]

2.2.5 Warenausgabe

In den bisher betrachteten Graphen wurde lediglich spezifiziert, wie ein bestimmter Geldbetrag durch eine Folge einzuwerfender Münzen entrichtet werden kann. Ein nützlicher Automat sollte allerdings nicht nur Münzen konsumieren, sondern anschließend auch eine bestimmte Ware ausgeben. Beispielsweise würde man von einem Zigarettenautomaten erwarten, daß er nach Einwurf von 5 DM die Entnahme eines Päckchens Zigaretten erlaubt. Abbildung 2.20 zeigt dies für einige fiktive Zigarettenmarken.

2.2.5.1 Wiederholung

Ein realistischer Zigarettenautomat sollte die Folge Fünf Mark − Päckchen Zigaretten allerdings nicht nur einmal, sondern idealerweise beliebig oft durchlaufen können. Außerdem soll beschrieben werden, daß der Automat zuerst einmal aufgestellt werden muß und eines Tages möglicherweise auch wieder abgebaut wird.

Abbildung 2.21 zeigt einen Interaktionsgraphen, der die zulässigen Ausführungsreihenfolgen beschreibt: Nach Passieren der Aktion Aufstellen trifft man auf eine *Wiederholung*, die wie folgt durchlaufen wird:

- Am linken ○-Knoten kann man, ähnlich wie bei einer Entweder-oder-Verzweigung, entweder nach unten abbiegen und den *Rumpf* der Wiederholung (d. h. den Teilgraphen zwischen den beiden ○-Knoten) umgehen oder aber nach rechts weitergehen und den Rumpf durchlaufen.

[5] Der Verzicht auf Rekursion stellt eine *bewußte* Einschränkung gegenüber Syntaxdiagrammen bzw. kontextfreien Grammatiken dar (vgl. § 2.8.2.5).

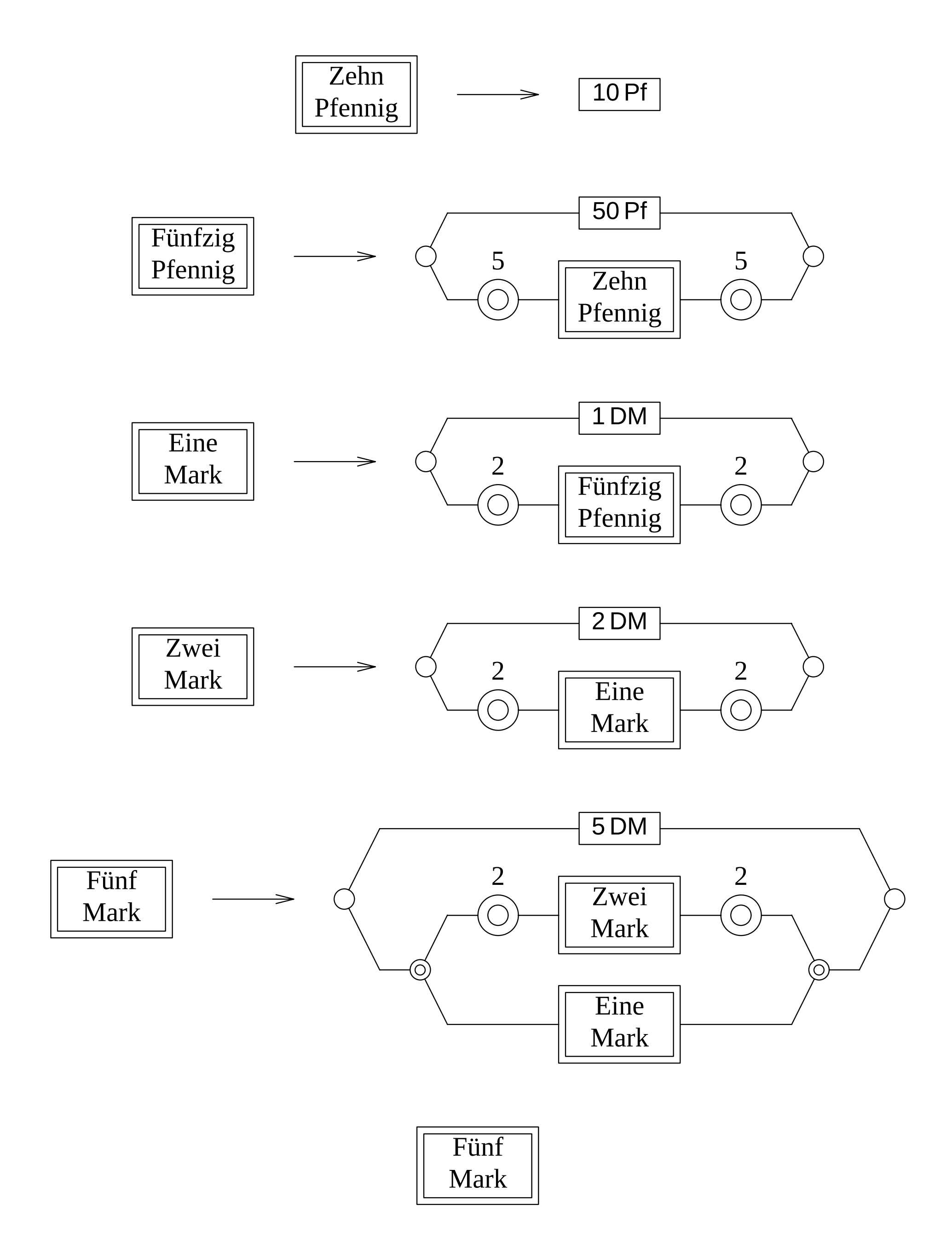


Abbildung 2.19: Einwerfen von 5 DM (Variante 7)

- Am rechten ○-Knoten hat man, wiederum vergleichbar einer Entweder-oder-Verzweigung, die Möglichkeit, nach rechts weiterzugehen und so die Wiederholung zu beenden oder aber nach oben abzubiegen und zum linken ○-Knoten zurückzukehren, an dem das „Spiel“ von neuem beginnt.

Auf diese Weise kann der Rumpf einer Wiederholung *beliebig oft* (ggf. auch keinmal) durchlaufen werden. Ein *Beenden* der Wiederholung ist jedoch nur möglich, wenn man sich am rechten ○-Knoten befindet, d. h. wenn man den zuletzt begonnenen „Iterationsschritt“ auch vollständig beendet hat. Im konkreten Beispiel bedeutet das, daß der Automat nur abgebaut werden kann, wenn gerade kein „Verkaufsvorgang“ aktiv ist.

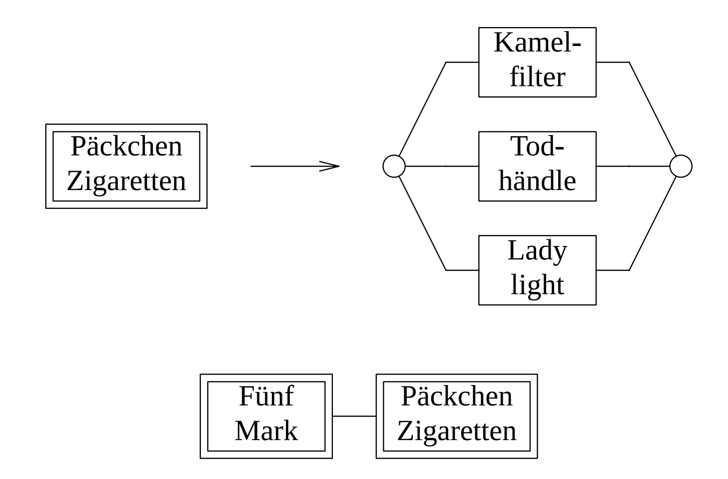


Abbildung 2.20: Einwurf von 5 DM und Ausgabe eines Päckchens Zigaretten

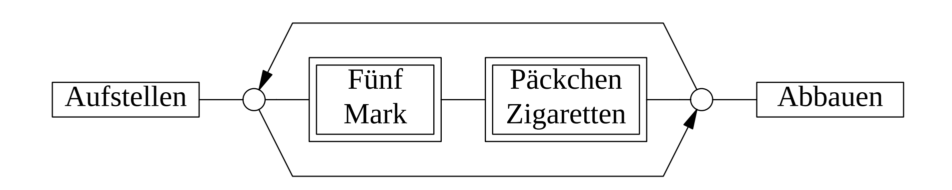


Abbildung 2.21: Zigarettenautomat

2.2.5.2 Eventuell-Verzweigung

Läßt man bei einer Wiederholung den „Rückweg" vom rechten zum linken ○-Knoten weg, so erhält man eine *Eventuell-Verzweigung* (vgl. Abb. 2.22), die besagt, daß der Teilgraph x entweder übersprungen oder *einmal* durchlaufen werden kann. Abbildung 2.23 zeigt als mögliche Anwendung einen „freundlichen" Zigarettenautomaten, der nach der Ausgabe eines Päckchens Zigaretten auch noch gratis die Entnahme eines Päckchens Zündhölzer gestattet.

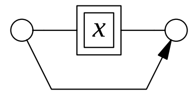


Abbildung 2.22: Eventuell-Verzweigung

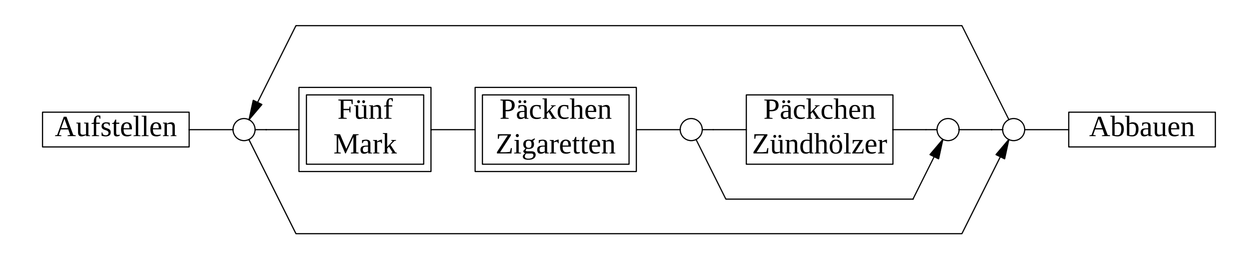


Abbildung 2.23: Freundlicher Zigarettenautomat

2.2.5.3 Mehrfach-Ausführung

Läßt man anstelle des „Rückwegs“ die untere „Umgehungsstraße“ vom linken zum rechten ○-Knoten einer Wiederholung weg und versieht die beiden Knoten außerdem mit einem Wiederholungsfaktor $n \in \mathbb{N}$, so erhält man eine *Mehrfach-Ausführung* (vgl. Abb. 2.24), die besagt, daß der Teilgraph x genau n-mal nacheinander durchlaufen werden muß. Der Graph in Abb. 2.25 beschreibt daher einen Zigarettenautomaten, der genau 100 Päckchen Zigaretten ausgeben kann.

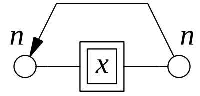


Abbildung 2.24: Mehrfach-Ausführung

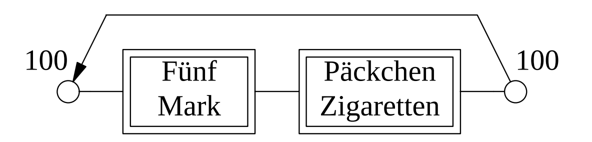


Abbildung 2.25: Zigarettenautomat mit 100 Päckchen Zigaretten

Anmerkung: Obwohl der Knotentyp ○, der zunächst zur Darstellung der Entweder-oder-Verzweigung eingeführt wurde, auch zur Formulierung der Wiederholung, der Eventuell-Verzweigung und der Mehrfach-Ausführung verwendet wird, darf hieraus nicht gefolgert werden, daß er zur Konstruktion beliebiger Verzweigungsstrukturen − wie man sie beispielsweise von Syntax- oder Flußdiagrammen kennt − verwendet werden darf. Vielmehr besitzen Interaktionsgraphen stets eine *symmetrische Blockstruktur*, in der es zu jedem Verzweigungsknoten einen zugehörigen Vereinigungsknoten gibt und umgekehrt. Insbesondere besitzt jeder Graph genau einen *Eingang* (ganz links) und einen *Ausgang* (ganz rechts).

2.2.6 Zusammenfassung

Tabelle 2.26 faßt die bisher eingeführten Operatoren von Interaktionsgraphen noch einmal zusammen, bevor im nachfolgenden Abschnitt 2.3 − anhand eines verwandten Beispiels − weiterführende Operatoren vorgestellt werden.

2.3 Weiterführende Operatoren (Beispiel Münzkopierer)

2.3.1 Normalbetrieb

2.3.1.1 Einfache Version

Ein einfacher Münzkopierer funktioniert prinzipiell ähnlich wie ein Zigarettenautomat: Man wirft einen bestimmten Geldbetrag ein, z. B. 10 Pfennig, und erhält dafür eine entsprechende „Ware“, beispielsweise eine DIN-A4-Fotokopie. Der Graph in Abb. 2.27 beschreibt dieses vereinfachte Verhalten.

Operator	*Abschnitt*
Entweder-oder-Verzweigung	2.2.1
Sowohl-als-auch-Verzweigung	2.2.2.4
Mehrfach-Verzweigung	2.2.3.3
Abkürzungen	2.2.4.4
Wiederholung	2.2.5.1
Eventuell-Verzweigung	2.2.5.2
Mehrfach-Ausführung	2.2.5.3

Tabelle 2.26: Grundlegende Operatoren von Interaktionsgraphen

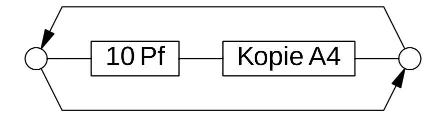


Abbildung 2.27: Münzkopierer (einfache Version)

2.3.1.2 Verbesserte Version

Abgesehen davon, daß ein echter Kopierer in der Regel auch „größere“ Münzen akzeptiert, erlaubt er normalerweise auch, daß man mehrere Münzen hintereinander einwirft und anschließend entsprechend oft hintereinander kopiert. Aber auch ein Nachwerfen von Münzen, bevor das bis jetzt eingeworfene Geld vollständig verbraucht ist, wird gewöhnlich unterstützt. Das bedeutet, daß die Aktionen 10 Pf und Kopie A4 in relativ beliebigen Reihenfolgen ausgeführt werden dürfen, sofern sichergestellt ist, daß die Anzahl der Kopien die Anzahl der eingeworfenen Groschen nicht übersteigt.

Um diese Bedingung zu formulieren, wird die Wiederholung in Abb. 2.27, die eine strikt alternierende Ausführung der Aktionen 10 Pf und Kopie A4 erzwingt, durch eine *Beliebig-oft-Verzweigung* ersetzt (Abb. 2.28), die besagt, daß die Sequenz 10 Pf – Kopie A4 *beliebig oft parallel* durchlaufen werden darf.

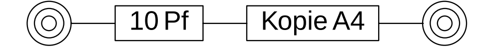


Abbildung 2.28: Münzkopierer (verbesserte Version)

2.3.1.3 Beliebig-oft-Verzweigung

Anschaulich bedeutet das, daß der Graph in Abb. 2.28 wie folgt durchlaufen wird:

- Ähnlich wie bei der Mehrfach-Verzweigung in § 2.2.3.3, stelle man sich eine *Mannschaft* von Läufern vor, die sich am linken ◎-Knoten in *beliebig viele* Gruppen aufteilt.
- Jede dieser Gruppen durchläuft *unabhängig* von den anderen Gruppen den Rumpf der Verzweigung, d. h. den Teilgraphen zwischen den beiden ◎-Knoten.
- Am rechten ◎-Knoten treffen die Gruppen, ebenso wie bei der Mehrfach-Verzweigung, wieder zusammen und setzen ihre Reise gemeinsam fort, sobald die Mannschaft komplett ist, d. h. alle Gruppen eingetroffen sind.

- Als Grenzfall ist es auch zulässig, daß die Mannschaft als ganzes vom linken zum rechten ◎-Knoten „springt", d. h. den Rumpf der Verzweigung komplett umgeht (ebenso wie der Rumpf einer Wiederholung komplett umgangen werden kann).
  Will man diese Möglichkeit auch graphisch zum Ausdruck bringen, so kann man zur Verdeutlichung eine zusätzliche „Umgehungsstraße" einzeichnen, die die beiden ◎-Knoten direkt miteinander verbindet (vgl. Abb. 2.29). Aus Symmetriegründen wird diese Kante jedoch normalerweise weggelassen.

Gemäß dieser Beschreibung kann eine Beliebig-oft-Verzweigung „beliebig oft x" auch als eine Mehrfach-Verzweigung „n-mal x" aufgefaßt werden, bei der der Verzweigungsfaktor n variabel ist, d. h. einen beliebigen Wert aus der Menge $\mathbb{N}_0$ annehmen kann.

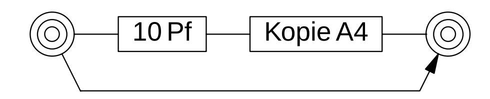


Abbildung 2.29: Beliebig-oft-Verzweigung mit expliziter „Umgehungsstraße"

Abhängig davon, wieviele Gruppen den Teilgraphen 10 Pf – Kopie A4 durchlaufen, können sich die folgenden Durchlaufreihenfolgen ergeben:

0. Keine Gruppe durchläuft den Rumpf der Verzweigung, d. h. die ganze Mannschaft springt vom linken ◎-Knoten direkt zum rechten.
   In diesem Fall erhält man eine leere Folge von Aktionen.

1. Genau eine Gruppe durchläuft den Teilgraphen 10 Pf – Kopie A4.
   In diesem Fall ergibt sich genau die Folge 10 Pf – Kopie A4.

2. Zwei Gruppen G_1 und G_2 durchlaufen unabhängig voneinander den Teilgraphen 10 Pf – Kopie A4. Sei (o. B. d. A.) G_1 diejenige Gruppe, die als erste die Aktion 10 Pf passiert. Dann können sich die folgenden Fälle ergeben:
   a) G_2 passiert 10 Pf, bevor G_1 die Aktion Kopie A4 passiert.
      Jetzt müssen beide Gruppen noch Kopie A4 passieren, um den rechten ◎-Knoten zu erreichen.
      Die resultierende Folge ist daher 10 Pf – 10 Pf – Kopie A4 – Kopie A4.
   b) G_1 passiert Kopie A4, bevor G_2 die Aktion 10 Pf passiert.
      Jetzt muß G_2 noch 10 Pf und Kopie A4 passieren, um ebenfalls den rechten ◎-Knoten zu erreichen.
      Resultat: 10 Pf – Kopie A4 – 10 Pf – Kopie A4.

3. Drei Gruppen durchlaufen unabhängig voneinander den Teilgraphen 10 Pf – Kopie A4.
   Ähnlich wie im Fall 2, können sich hier die folgenden Reihenfolgen ergeben:
   a) 10 Pf – 10 Pf – 10 Pf – Kopie A4 – Kopie A4 – Kopie A4
      (Alle drei Gruppen passieren 10 Pf, bevor die erste Gruppe Kopie A4 passiert.)
   b) 10 Pf – 10 Pf – Kopie A4 – 10 Pf – Kopie A4 – Kopie A4
      (Zwei Gruppen passieren 10 Pf und eine von ihnen Kopie A4, bevor die dritte Gruppe 10 Pf passiert.)

c) 10 Pf – 10 Pf – Kopie A4 – Kopie A4 – 10 Pf – Kopie A4
(Zwei Gruppen passieren 10 Pf und anschließend beide Kopie A4, bevor die dritte Gruppe 10 Pf und Kopie A4 passiert.)

d) 10 Pf – Kopie A4 – 10 Pf – 10 Pf – Kopie A4 – Kopie A4
(Eine Gruppe passiert 10 Pf und Kopie A4, bevor die anderen beiden Gruppen 10 Pf und anschließend Kopie A4 passieren.)

e) 10 Pf – Kopie A4 – 10 Pf – Kopie A4 – 10 Pf – Kopie A4
(Die drei Gruppen durchlaufen die Sequenz 10 Pf – Kopie A4 strikt sequentiell.)

4. Und so weiter.

Ähnlich wie bei einer Sowohl-als-auch-Verzweigung oder bei einer Mehrfach-Verzweigung, erhält man die zulässigen Ausführungsreihenfolgen also, indem man beliebig viele Folgen 10 Pf – Kopie A4 miteinander verschränkt. Zur Veranschaulichung der Verschränkung von *n* Folgen stelle man sich eine *n*-spurige Autobahn vor, die sich an einer Baustelle auf eine Spur verengt. Auf jeder der *n* Spuren steht eine Schlange von Fahrzeugen, die sich nun nach einem beliebigen Schema auf eine einzige Spur fädeln und so die Baustelle passieren. Wenn auf jeder der *n* Spuren jeweils ein Fahrzeug mit der Aufschrift 10 Pf und dahinter eines mit der Aufschrift Kopie A4 steht, kann letzteres frühestens dann einfädeln, wenn sein „Vordermann" eingefädelt hat. Somit haben zu jedem Zeitpunkt mindestens so viele 10 Pf-Autos wie Kopie A4-Autos die Engstelle passiert. Übertragen auf Folgen von Aktionen bedeutet das, daß zu jedem Zeitpunkt mindestens so viele 10-Pfennig-Münzen eingeworfen wie Kopien erstellt wurden.

2.3.1.4 Komfort-Version

Abbildung 2.30 beschreibt eine weitere Verbesserung des Münzkopierers aus Abb. 2.28: Nach Einwurf einer 10-Pfennig-Münze kann man entweder eine A4-Kopie erstellen (oberer Zweig der Entweder-oder-Verzweigung) oder aber eine weitere 10-Pfennig-Münze einwerfen und dann eine A3-Kopie erstellen (unterer Zweig). Durch die umgebende Beliebig-oft-Verzweigung können wiederum beliebig viele dieser Sequenzen gleichzeitig aktiv sein, was interessante Kombinationsmöglichkeiten eröffnet:

- Wenn beispielsweise *eine* Gruppe den Rumpf der Beliebig-oft-Verzweigung durchläuft und am Entweder-oder-Verzweigungsknoten nach *unten* abbiegt, erhält man die Aktionsfolge

  10 Pf – 10 Pf – Kopie A3,

  d. h. man kann nach Einwurf von zwei Groschen eine A3-Kopie erstellen.

- Wird der Rumpf von *zwei* Gruppen durchlaufen, die beide den *oberen* Zweig der Entweder-oder-Verzweigung wählen, so kann sich unter anderem die Folge

  10 Pf – 10 Pf – Kopie A4 – Kopie A4

  ergeben, d. h. nach Einwurf von zwei Groschen kann man auch zwei A4-Kopien erstellen.

- Wenn eine Gruppe den oberen und die andere den unteren Zweig durchläuft, erhält man u. a. die Folgen

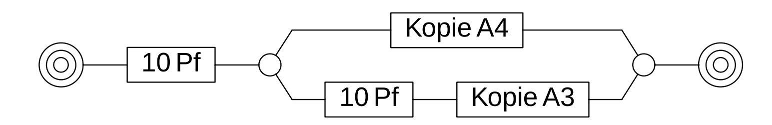


Abbildung 2.30: Münzkopierer (Komfort-Version)

10 Pf – 10 Pf – 10 Pf – Kopie A4 – Kopie A3 und

10 Pf – 10 Pf – 10 Pf – Kopie A3 – Kopie A4,

d. h. nach Einwurf von drei Groschen kann man z. B. eine A4- und eine A3-Kopie (in beliebiger Reihenfolge) erstellen.

- Da die beiden Gruppen den Rumpf der Beliebig-oft-Verzweigung unabhängig voneinander durchlaufen, sind aber auch die Reihenfolgen

  10 Pf – Kopie A4 – 10 Pf – 10 Pf – Kopie A3,

  10 Pf – 10 Pf – Kopie A4 – 10 Pf – Kopie A3 und

  10 Pf – 10 Pf – Kopie A3 – 10 Pf – Kopie A4

  möglich usw.

2.3.2 Papier nachfüllen

2.3.2.1 Wechselseitiger Ausschluß

Von Zeit zu Zeit muß bei einem Kopierer Papier nachgefüllt werden, indem eines seiner Papierfächer geöffnet, Papier des entsprechenden Formats eingelegt und anschließend das Papierfach wieder geschlossen wird. Um nun z. B. zu beschreiben, daß bei geöffnetem A4-Papierfach keine A4-Kopie erstellt werden kann, muß spezifiziert werden, daß sich die Aktionen bzw. Aktionsfolgen Kopie A4 und Öffnen A4 – ... – Schließen A4 *wechselseitig ausschließen*, d. h. daß zu jedem Zeitpunkt höchstens *eine* dieser Folgen durchlaufen werden kann.

Abbildung 2.31 zeigt einen entsprechenden Graphen: Die Entweder-oder-Verzweigung im Rumpf der Wiederholung darf beliebig oft durchlaufen werden, bei jedem Durchlauf muß man sich aber für einen der beiden Zweige entscheiden, d. h. man kann zu jedem Zeitpunkt *entweder* kopieren *oder* aber Papier nachfüllen.

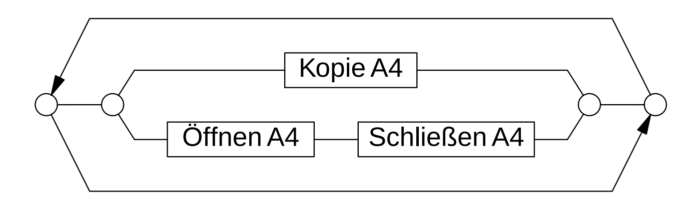


Abbildung 2.31: Integritätsbedingung für Papierformat A4

Man beachte, daß dieser Graph nur die zur Formulierung der gewünschten Integritätsbedingung *unbedingt erforderlichen* Aktionen Kopie A4, Öffnen A4 und Schließen A4 enthält, über die konkreten Tätigkeiten *zwischen* dem Öffnen und Schließen des Papierfachs jedoch *keine Aussage* macht, weil sie an dieser Stelle nicht relevant sind. (Abgesehen davon könnte es tatsächlich vorkommen, daß die Aktionen Öffnen A4 und Schließen A4 *unmittelbar nacheinander* ausgeführt werden, wenn man nach dem Öffnen des Papierfachs feststellt, daß noch genügend Papier eingelegt ist.) Außerdem macht der Graph *keine Aussage* über das Einwerfen von Münzen, weil dieser Aspekt eines Münzkopierers bereits durch den Graphen in Abb. 2.30 abgedeckt wird. Und schließlich wird an dieser Stelle nur die Integritätsbedingung für das Papierformat A4 formuliert, während der Graph in Abb. 2.32 − *unabhängig davon* − die entsprechende Bedingung für das Format A3 spezifiziert.

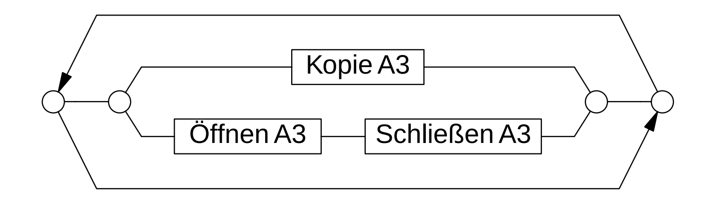


Abbildung 2.32: Integritätsbedingung für Papierformat A3

2.3.2.2 Kopplung

Um nun auszudrücken, daß die Bedingungen der Abbildungen 2.30, 2.31 und 2.32 *zusammen* eingehalten werden müssen, d. h. daß nur Ausführungsreihenfolgen zulässig sind, die von allen drei Graphen akzeptiert werden, werden sie in Abb. 2.33 mit Hilfe einer *Kopplung* (o· · ·o) verknüpft, die anschaulich wie folgt durchlaufen wird:

- Die einzelnen Zweige der Kopplung werden, ähnlich wie die einer Sowohl-als-auch-Verzweigung, *parallel* durchlaufen.
- Im Gegensatz zu einer Sowohl-als-auch-Verzweigung dürfen die Zweige jedoch nur *teilweise* unabhängig voneinander durchlaufen werden:
  - Aktionen, die nur in *einem* Zweig der Kopplung vorkommen (wie z. B. 10 Pf oder Öffnen A4), können in diesem Zweig *unabhängig* von den anderen Zweigen durchlaufen werden.
  - Aktionen, die in *zwei* Zweigen auftreten (wie z. B. Kopie A4 und Kopie A3), dürfen nur durchlaufen werden, wenn sie in beiden Zweigen *gleichzeitig* passiert werden können, wobei die entsprechende Aktion in der „realen Welt“ nur *einmal* ausgeführt wird.
  - Allgemein dürfen Aktionen, die in *n* Zweigen der Kopplung auftreten, nur durchlaufen werden, wenn sie in diesen *n* Zweigen gleichzeitig passiert werden können.

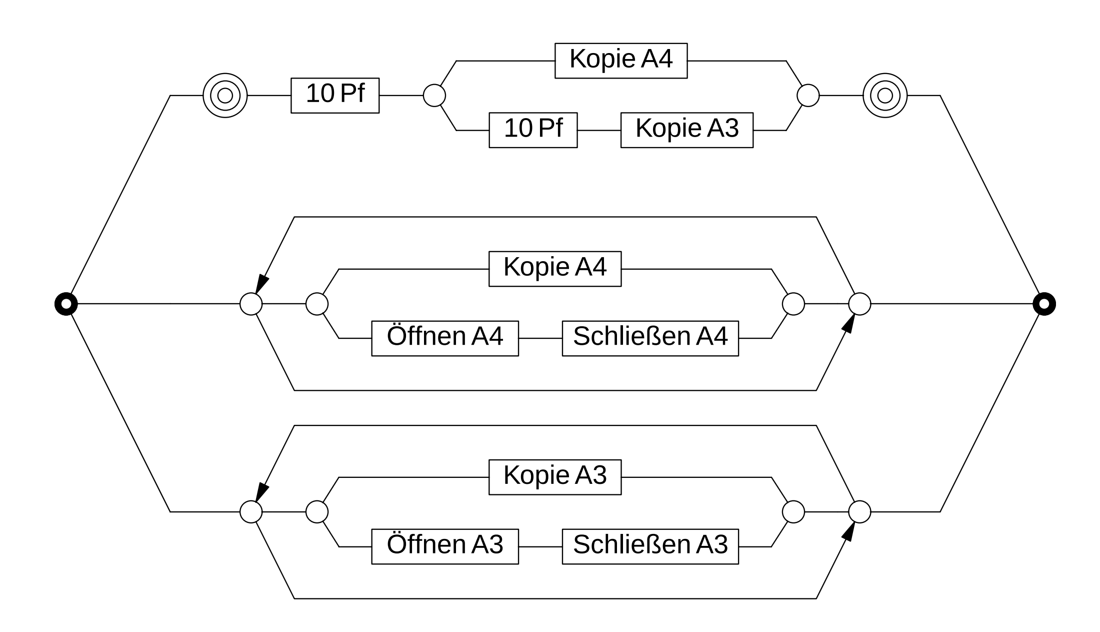


Abbildung 2.33: Kopplung von Bedingungen

- Am Vereinigungsknoten ⭘ (rechts) kann die Reise fortgesetzt werden, sobald alle Zweige vollständig durchlaufen sind.

Durch die genannten Regeln wird sichergestellt, daß eine Aktion nur ausgeführt werden kann, wenn sie von *allen* Zweigen der Kopplung, die über diese Aktion *eine Aussage machen*, zugelassen wird. Zweige, die über die fragliche Aktion *keine* Aussage machen, müssen jedoch nicht berücksichtigt werden, d. h. sie lassen die Aktion jederzeit zu.

Anmerkung: Ebenso wie bei allen anderen Arten von Interaktionsgraphen, können aber grundsätzlich nur Aktionen durchlaufen werden, die im Graphen tatsächlich vorkommen. Das bedeutet, daß Aktionen, die in *keinem* Zweig der Kopplung auftreten, *nie* zugelassen werden. (Vgl. jedoch § 5.2.6.)

Anmerkung zur graphischen Darstellung: Das Symbol ⭘ zur Darstellung der Kopplung kann als modifiziertes ◎-Symbol (Sowohl-als-auch-Verzweigung) aufgefaßt werden, bei dem die teilweise Schwärzung andeutet, daß die einzelnen Zweige teilweise miteinander gekoppelt sind. Entsprechend gibt es auch ein Symbol ●, dessen vollständige Schwärzung anzeigt, daß seine Zweige vollständig oder *strikt* gekoppelt traversiert werden müssen, d. h. daß eine Aktion nur dann ausgeführt werden darf, wenn sie in *allen* Zweigen gleichzeitig passiert werden kann. Die praktische Bedeutung dieses Operators ist jedoch äußerst gering.

2.3.2.3 Beispiel

Der Graph in Abb. 2.33 kann *zum Beispiel* wie folgt durchlaufen werden:

1. Am linken ⭘-Knoten beginnt man, alle drei Zweige der Kopplung zu durchlaufen.
   Stellt man sich zum Durchlaufen des oberen Zweigs wieder eine Mannschaft von Läufern vor, so teilt sich diese am linken ◎-Knoten *zum Beispiel* in zwei Gruppen G_1 und G_2 auf, die beide zunächst die (linke) Aktion 10 Pf durchlaufen. Anschließend biegt eine der Gruppen (z. B. G_1) am Entweder-oder-Verzweigungsknoten nach unten ab und durchläuft die dort folgende Aktion 10 Pf, während die andere Gruppe G_2 nach oben abbiegt. Wie bereits erwähnt, können die Aktionen 10 Pf jeweils unabhängig von den übrigen Zweigen der Kopplung passiert werden, weil sie nur in diesem Zweig auftreten.
   In der realen Welt wurde jetzt dreimal die Aktion 10 Pf ausgeführt, d. h. ein Kunde hat drei 10-Pfennig-Münzen in den Kopierer eingeworfen.

2. Im mittleren Zweig der Kopplung betritt man den Rumpf der Wiederholung, biegt am nachfolgenden Entweder-oder-Verzweigungsknoten *zum Beispiel* nach unten ab und durchläuft die Aktion Öffnen A4. Auch dies ist unabhängig von den übrigen Zweigen der Kopplung möglich.
   In der Realität hat jetzt z. B. ein Angestellter des Kopiergeschäfts das A4-Papierfach geöffnet, um Papier nachzufüllen.

3. Da die Aktion Kopie A4 im oberen *und* mittleren Zweig der Kopplung auftritt, in letzterem aber momentan nicht durchlaufen werden kann (weil sich der Läufer dieses Zweigs gerade zwischen den Aktionen Öffnen A4 und Schließen A4 befindet), darf sie auch im oberen Zweig nicht passiert werden, d. h. momentan kann keine A4-Kopie erstellt werden.
   Ein gleichzeitiges Durchlaufen der Aktion Kopie A3 im oberen und unteren Zweig der Kopplung ist jedoch möglich, weil die oben erwähnte Gruppe G_1 diese Aktion passieren kann und der Läufer des unteren Kopplungszweigs den Rumpf „seiner“ Wiederholung betreten, am Entweder-oder-Verzweigungsknoten nach oben abbiegen und somit ebenfalls die Aktion Kopie A3 durchlaufen kann.
   In der realen Welt wurde jetzt *einmal* die Aktion Kopie A3 ausgeführt, d. h. der Kunde hat eine A3-Kopie erstellt.

4. Im mittleren Zweig der Kopplung kann jetzt z. B. − unabhängig von den übrigen Zweigen − die Aktion Schließen A4 durchlaufen werden. Anschließend erreicht man dort den Entweder-oder-Vereinigungsknoten sowie den Endknoten der Wiederholung, an dem man zu ihrem Anfangsknoten

zurückkehren und den Rumpf erneut betreten kann.
In der Realität hat der Angestellte des Kopiergeschäfts jetzt das A4-Papierfach wieder geschlossen.

5. Wählt man im mittleren Zweig der Kopplung nun den oberen Zweig der Entweder-oder-Verzweigung, so kann die Aktion Kopie A4 jetzt im oberen und mittleren **o**-Zweig gleichzeitig durchlaufen werden, weil die oben erwähnte Gruppe G_2 ebenfalls in der Lage ist, diese Aktion zu durchlaufen.
In der realen Welt hat der Kunde jetzt eine A4-Kopie erstellt und damit sein bisher eingeworfenes Geld verbraucht.

Anmerkung: Im Verlauf dieses Beispiels wurden immer wieder *willkürliche* Entscheidungen getroffen: Die Mannschaft im oberen Zweig der Kopplung teilt sich in zwei Gruppen auf, eine dieser Gruppen durchläuft den unteren Entweder-oder-Zweig, die andere den oberen usw. Hätte man andere Entscheidungen getroffen, so hätte man (mit großer Wahrscheinlichkeit) eine *andere* Folge von passierten Aktionen erhalten. Wollte man für eine *vorgegebene* Folge von Aktionen überprüfen, ob sie aus Sicht des Graphen zulässig ist, so wäre es ratsam, den Graphen systematisch und *zielgerichtet* zu durchlaufen. Dies wird in § 2.4 genauer erläutert.

2.3.2.4 Modulare Kombination

Die speziellen Regeln zur Traversierung einer Kopplung, die auf den ersten Blick vielleicht etwas umständlich wirken mögen, erlauben es − wie in diesem Abschnitt erläutert − *Teilbedingungen* einer umfassenderen Integritätsbedingung *unabhängig voneinander* zu entwickeln und anschließend mit Hilfe einer Kopplung zu einer Gesamtbedingung zusammenzufügen. Da die einzelnen Teilbedingungen hierbei *unverändert* übernommen werden können, könnte ihre Spezifikation beispielsweise auch in einer Abkürzung *verborgen* sein. Diese Vorgehensweise, die beim praktischen Einsatz von Interaktionsgraphen sehr häufig angewandt wird, wird als *modulare Kombination* von Graphen bezeichnet.

2.3.3 Schablonen

2.3.3.1 Wechselseitiger Ausschluß

Den beiden Graphen in Abb. 2.31 und 2.32 liegt ein häufig wiederkehrendes syntaktisches *Muster* zugrunde: Zwei (oder mehr) Teilgraphen werden mit Hilfe einer Entweder-oder-Verzweigung verknüpft, die ihrerseits in eine Wiederholung eingebettet ist. Semantisch assoziiert man mit diesem Muster das *Konzept* des wechselseitigen Ausschlusses, d. h. man könnte die Konstruktion „Wiederholung einer Entweder-oder-Verzweigung“ als eine (von möglicherweise mehreren verschiedenen) *Implementierungen* des abstrakten Konzepts „wechselseitiger Ausschluß von Zweigen“ betrachten. (Tatsächlich sind andere Implementierungsformen desselben Konzepts denkbar, wie Abb. 2.34 zeigt.)

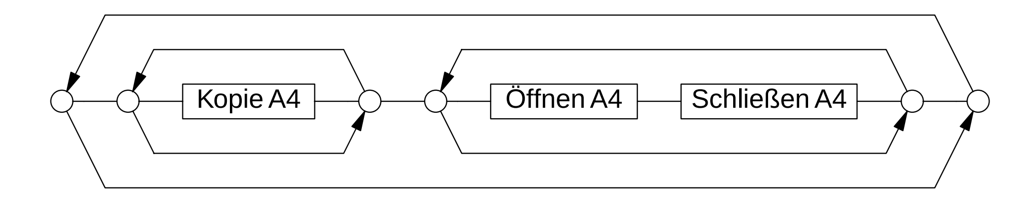


Abbildung 2.34: Alternative Implementierung eines wechselseitigen Ausschlusses

Mit Hilfe einer *Schablone* kann dieser Abstraktionsschritt auch graphisch zum Ausdruck gebracht werden: In Abb. 2.35 wird mit Hilfe einer „Blitz-Verzweigung“ auf einer abstrakten Ebene spezifiziert, daß sich die Ausführung der Zweige Kopie A4 und Öffnen A4 − Schließen A4 zeitlich nicht über-

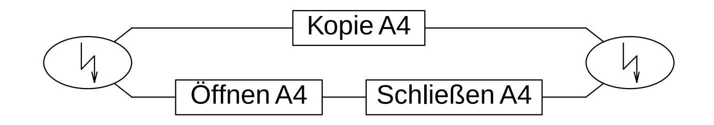


Abbildung 2.35: Wechselseitiger Ausschluß mit Schablone

lappen darf. Wie diese allgemeine Bedingung konkret mit Hilfe elementarer Interaktionsgraphen formuliert werden kann, wird separat − und vor allem nur ein einziges Mal − in Abb. 2.36 spezifiziert.

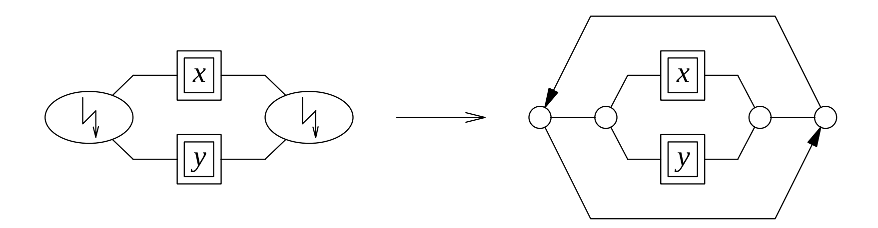


Abbildung 2.36: Definition der Schablone mutex mit zwei Zweigen

Syntaktisch ähnelt eine Schablone einer Verzweigung: Sie besitzt einen Verzweigungsknoten zur linken und einen Vereinigungsknoten zur rechten, zwischen denen sich prinzipiell beliebig viele Teilgraphen befinden können. Als Verzweigungs- und Vereinigungsknoten dienen Ellipsen, die entweder mit dem *Namen* der Schablone (wie z. B. mutex für mutual exclusion) oder einem „sprechenden" graphischen *Symbol* (im Beispiel ein Blitz) beschriftet werden.

Konzeptuell entspricht eine Schablone einer *parametrisierten Abkürzung*. Dementsprechend beschreibt die Definition der Schablone in Abb. 2.36, daß eine Blitz-Verzweigung mit den Zweigen x und y eine Abkürzung einer Wiederholung über eine Entweder-oder-Verzweigung „x oder y" darstellt. Da die *formalen Parameter* der Schablone, x und y, Platzhalter für beliebig komplexe Teilgraphen darstellen, werden sie − ebenso wie Abkürzungen − mit Doppelrahmen umgeben.

2.3.3.2 Variable Schablonen

Die soeben vorgestellte Definition der Schablone mutex ist insofern unpraktisch, als sie genau *zwei* Zweige besitzt und daher nur zur Formulierung des wechselseitigen Ausschlusses von genau zwei Teilgraphen verwendet werden kann. Abbildung 2.37 hingegen zeigt eine flexiblere Definition, die besagt, daß eine Blitz-Verzweigung mit *beliebig vielen* Zweigen $x, \ldots$ eine Abkürzung einer Wiederholung über eine Entweder-oder-Verzweigung „x oder $\ldots$" darstellt. Mit Hilfe dieser Schablone können

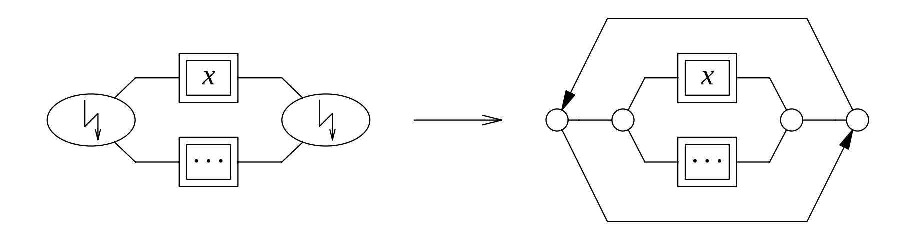


Abbildung 2.37: Definition der Schablone mutex mit beliebig vielen Zweigen

daher wechselseitige Ausschlußbedingungen mit beliebig vielen Teilgraphen $x_1, \ldots, x_n$ formuliert werden, die für eine Wiederholung einer Entweder-oder-Verzweigung „x_1 oder … oder x_n“ stehen.

Unter Verwendung dieser Schablonendefinition kann der Graph aus Abb. 2.33 (§ 2.3.2.2) nun kompakter, übersichtlicher und vor allem auf höherem Abstraktionsniveau gemäß Abb. 2.38 dargestellt werden.

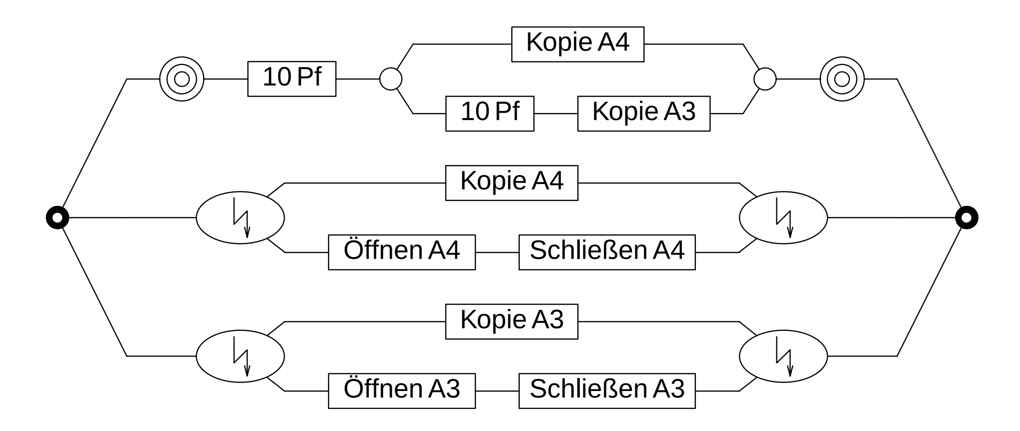


Abbildung 2.38: Münzkopierer mit Schablonen

2.3.4 Mehrere Kopierer

2.3.4.1 Erster Versuch

Da es in einem Kopiergeschäft in der Regel nicht nur einen, sondern mehrere (beispielsweise zehn) Kopierer gibt, die grundsätzlich unabhängig voneinander funktionieren, ist man möglicherweise geneigt, die Integritätsbedingung für dieses Gesamtsystem mit Hilfe einer Mehrfach-Verzweigung gemäß Abb. 2.39 zu formulieren. (Zur Vereinfachung wurden die beiden Nebenbedingungen aus Abb. 2.38 wieder weggelassen.)

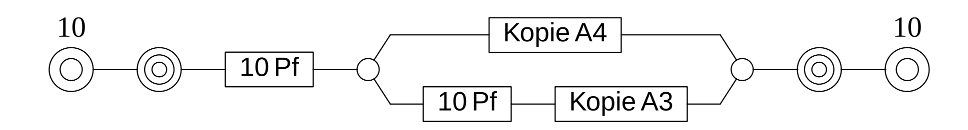


Abbildung 2.39: Kopiergeschäft mit zehn Kopierern (erster Versuch)

Da die Aktionen in diesem Graphen jedoch nicht *Kopierer-spezifisch* formuliert sind, kann man grundsätzlich nicht unterscheiden, in welchen Kopierer z. B. gerade eine Münze eingeworfen oder mit welchem gerade eine Kopie erstellt wird. Folglich würde der Graph beispielsweise erlauben, mit Kopierer Nr. 7 eine A4-Kopie zu erstellen, nachdem man in Kopierer 3 eine 10-Pfennig-Münze eingeworfen hat!

2.3.4.2 Zweiter Versuch

Um dieses unerwünschte Verhalten verhindern zu können, ist es erforderlich, sämtliche Aktionen in Abb. 2.39 mit einem *Parameter* (z. B. k) zu versehen, der in geeigneter Art und Weise einen der zehn

Kopierer *identifiziert*. Verwendet man zur Bezeichnung der Kopierer konkret die natürlichen Zahlen von 1 bis 10, so kann die korrekte Integritätsbedingung wie in Abb. 2.40 formuliert werden:[6] Der Teilgraph *zwischen* den beiden ◎-Knoten beschreibt die gewünschte Bedingung für einen *einzelnen* Kopierer k, und mit Hilfe der Summennotation $\underset{k=1}{\overset{10}{◎}}$ wird im Prinzip eine Sowohl-als-auch-Verzweigung mit zehn derartigen Zweigen für $k = 1$ bis 10 formuliert.

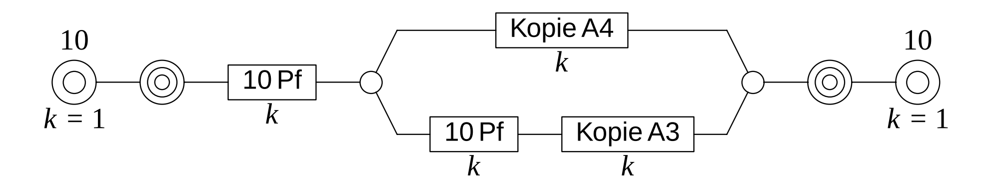


Abbildung 2.40: Kopiergeschäft mit zehn Kopierern (zweiter Versuch)

Der Graph wird also durchlaufen, indem man zehn verschiedene Ausprägungen des Teilgraphen mit den *Parameterbelegungen* $k = 1$ bis $k = 10$ unabhängig voneinander durchläuft. Der erste dieser Teilgraphen enthält somit nur Aktionen mit der Parameterbelegung $k = 1$, der zweite nur solche mit der Belegung $k = 2$ usw. Jeder Teilgraph sorgt daher − unabhängig von den anderen Teilgraphen − dafür, daß die Aktionen an „seinem" Kopierer in einer zulässigen Reihenfolge ausgeführt werden, und durch die Verknüpfung mittels einer Sowohl-als-auch-Verzweigung können Aktionen an verschiedenen Kopierern beliebig überlappend oder verschränkt ausgeführt werden.

2.3.4.3 Allgemeingültige Lösung

Die Zahl 10 im obigen Beispiel wurde absolut willkürlich gewählt, um einen konkreten Wert vor Augen zu haben; reale Kopiergeschäfte können natürlich sowohl mehr als auch weniger Kopiergeräte besitzen und − diese Anzahl ändert sich vermutlich von Zeit zu Zeit. Ebenso willkürlich ist die Annahme, daß die einzelnen Kopierer mit den natürlichen Zahlen von 1 bis 10 bezeichnet werden. Sinnvoller wäre vermutlich eine Identifikation über ihre Seriennummer, Inventarnummer oder ähnliches. Dann müßte man die Summenschreibweise $\underset{k=1}{\overset{10}{◎}}$ allerdings durch eine Notation der Art $\underset{k \in \Omega}{◎}$ ersetzen, in der die Indexmenge Ω die Nummern aller Kopierer des betrachteten Kopiergeschäfts enthält.

Dies ist jedoch aus verschiedenen Gründen unpraktisch: Zum einen ändert sich die Menge Ω immer wieder, wenn ein neuer Kopierer angeschafft oder ein alter ausgemustert wird. Zum anderen ist ihr konkreter Inhalt vom konkret betrachteten Kopiergeschäft abhängig, d. h. es ist auf diese Weise nicht möglich, eine *allgemeingültige* Integritätsbedingung zu formulieren, die auf beliebige Kopiergeschäfte angewandt werden kann. Schließlich ist es einfach lästig, wenn man in einer Inventarliste o. ä. die konkreten Nummern aller Kopiergeräte nachschlagen muß, wenn man eigentlich formulieren möchte, daß die Integritätsbedingung, die durch den Rumpf der ◎-Verzweigung beschrieben wird, *für alle* Kopierer eines Kopiergeschäfts gelten soll − unabhängig davon, wie ihre konkreten Nummern lauten.

Aus diesen Gründen ist es naheliegend, die Indexmenge Ω in der obigen Summenschreibweise einfach wegzulassen (vgl. Abb. 2.41) und so zum Ausdruck zu bringen, daß sie in konkreten Anwendungen meist weder genau bekannt noch von besonderem Interesse ist. Der Parameter k soll einfach (zumindest gedanklich) *alle prinzipiell denkbaren* Werte durchlaufen. Dementsprechend wird der Graph $\underset{k}{◎} \cdots \underset{k}{◎}$ auch als *Für-alle-Verzweigung* bezeichnet.

[6] Parameter von Aktionen werden entweder oberhalb oder unterhalb der zugehörigen Rechtecke plaziert.

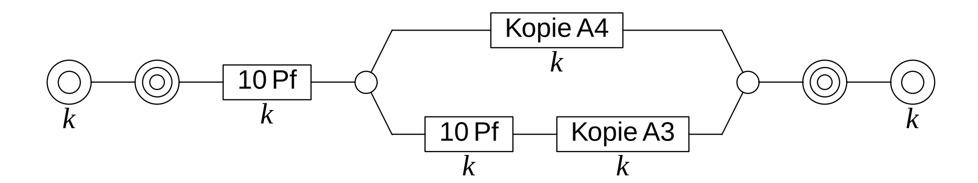


Abbildung 2.41: Kopiergeschäft mit beliebig vielen Kopierern

Anmerkung: Da der konkrete Wertebereich des Parameters *k* nun bewußt offengelassen wird, könnte *k* prinzipiell nicht nur Inventar- oder Seriennummern von Kopierern, sondern z. B. auch Personal- oder Kundennummern durchlaufen. Um dies zu vermeiden, müßte man Parameter *typisieren*, d. h. ihren Wertebereich auf eine bestimmte (endliche oder unendliche) Menge von Werten einschränken. Für die meisten praktischen Anwendungen von Interaktionsgraphen ist dies jedoch nicht erforderlich, da die Menge der potentiell ausführbaren Aktionen − und damit auch die Menge der möglichen Parameterbelegungen von Aktionen − anderweitig eingeschränkt wird. Beispielsweise kann die Aktion 10 Pfennig in Kopierer 080265H001 einwerfen, die der Graph in Abb. 2.41 prinzipiell akzeptieren würde, in der Realität niemals auftreten, wenn die Nummer 080265H001 kein Kopiergerät, sondern beispielsweise einen Kunden (mit Geburtsdatum 8.2.1965 und Initiale H) bezeichnet. Handelt es sich bei den zu synchronisierenden Aktionen um Workflowschritte, so sorgt darüber hinaus das verantwortliche WfMS dafür, daß nur Aktionen mit sinnvollen Parameterbelegungen zur Ausführung kommen können.

2.3.4.4 Für-alle-Verzweigungen

Eine Für-alle-Verzweigung wie in Abb. 2.41 wird ähnlich wie eine Beliebig-oft-Verzweigung durchlaufen:

- Am linken ◎-Knoten teilt sich die Mannschaft in so viele Gruppen auf, daß jeder Gruppe *eineindeutig* ein möglicher Wert des Parameters *k* zugeordnet werden kann.
- Die einzelnen Gruppen durchlaufen *unabhängig voneinander* den Rumpf der Verzweigung.
- Beim Durchlaufen einer *parametrisierten Aktion* wird deren Parameter *k* durch den Wert *ersetzt*, der der durchlaufenden Gruppe zugeordnet ist.
  In der realen Welt muß die Aktion daher mit *diesem* Parameterwert ausgeführt werden.
- Am rechten ◎-Knoten treffen die Gruppen wieder zusammen und setzen ihre Reise gemeinsam fort, sobald *alle* Gruppen eingetroffen sind.

Anmerkung: Da die Menge der prinzipiell möglichen Parameterwerte in vielen Anwendungen sehr groß oder sogar unbeschränkt ist, wird im folgenden grundsätzlich von einer *unendlichen Menge* ausgegangen. Dies bedeutet, daß eine Für-alle-Verzweigung anschaulich einer Sowohl-als-auch-Verzweigung mit *unendlich vielen Zweigen* entspricht. Auf die möglichen Konsequenzen dieser Tatsache wird in § 2.5.2 genauer eingegangen.

2.3.4.5 Beispiel

Konkret kann der Graph in Abb. 2.41 *zum Beispiel* wie folgt durchlaufen werden:

1. Am linken ◎-Knoten teilt sich die Gesamtmannschaft in unendlich viele Gruppen auf. Einer der Gruppen wird *zum Beispiel* die Seriennummer X5, einer anderen die Nummer U7 zugeordnet.
2. Die Gruppe (mit der Nummer) X5 teilt sich am Beliebig-oft-Verzweigungsknoten ◉ in zwei Teilgruppen, die beide die initiale Aktion 10 Pf mit der Parameterbelegung k = X5 durchlaufen.

Übertragen auf die reale Welt bedeutet das, daß in den Kopierer mit der Seriennummer X5 zwei Groschen eingeworfen wurden.

3. Eine der beiden X5-Teilgruppen geht weiter, biegt am Entweder-oder-Verzweigungsknoten nach oben ab und durchläuft die Aktion Kopie A4, ebenfalls mit der Parameterbelegung k = X5.
   Am Kopierer X5 wurde jetzt also eine A4-Kopie erstellt.
4. Die Gruppe U7 durchläuft als ganzes den Rumpf der Beliebig-oft-Verzweigung, d. h. sie „teilt“ sich quasi in eine einzige „Teilgruppe“ auf. Sie passiert die erste Aktion 10 Pf, biegt am Entweder-oder-Verzweigungsknoten nach unten ab, durchläuft die zweite Aktion 10 Pf und schließlich die Aktion Kopie A3, jeweils mit der Parameterbelegung k = U7.
   In der Realität bedeutet das, daß am Kopierer U7 zwei Groschen eingeworfen und eine A3-Kopie erstellt wurde.
5. Die zweite Teilgruppe der Gruppe X5 durchläuft ebenfalls die Aktion Kopie A4, d. h. am Kopierer X5 wird eine weitere A4-Kopie erstellt.
6. Da beide Teilgruppen der Gruppe X5 nun den Beliebig-oft-Vereinigungsknoten ◎ erreicht haben, kann die Gruppe X5 als ganzes nach rechts weitergehen und erreicht somit den rechten ◎-Knoten. Dasselbe gilt auch für die Gruppe U7.
7. Die übrigen unendlich vielen Gruppen, die am linken ◎-Knoten gebildet wurden (und dort irgendwelche nicht näher interessierenden Nummern zugeordnet bekamen), können den Rumpf der Beliebig-oft-Verzweigung komplett überspringen (vgl. § 2.3.1.3) und so direkt, ohne irgendeine Aktion zu passieren, den rechten ◎-Knoten erreichen.
8. Da somit *alle* unendlich vielen Gruppen den rechten ◎-Knoten erreicht haben, könnte die gesamte Mannschaft nach rechts weitergehen, wenn der Graph dort noch nicht zu Ende wäre.

2.3.4.6 Für-ein-Verzweigungen

Abbildung 2.42 zeigt einen letzten Basisoperator von Interaktionsgraphen, die *Für-ein-Verzweigung* $\underset{k}{\bigcirc}\cdots\underset{k}{\bigcirc}$. Ebenso wie eine Für-alle-Verzweigung im Prinzip eine unendliche *Sowohl-als-auch-Verzweigung* darstellt, entspricht eine Für-ein-Verzweigung einer *Entweder-oder-Verzweigung* mit unendlich vielen Zweigen, von denen genau *einer* zu durchlaufen ist. Anschaulich bedeutet das, daß man am linken ○-Knoten *irgendeinen* Wert für den Parameter k wählt, mit dem man den Rumpf der Verzweigung einmal durchläuft. Dementsprechend beschreibt der Graph in Abb. 2.42 einen Kunden, der ein Kopiergeschäft betritt, an irgendeinem der vorhandenen Kopierer seine Kopierarbeiten erledigt und anschließend das Geschäft wieder verläßt.

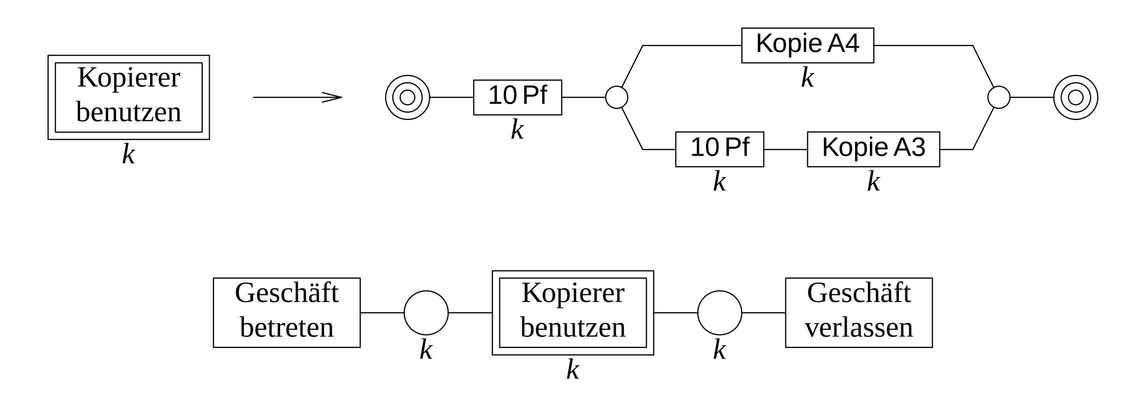


Abbildung 2.42: Für-ein-Verzweigung

Will man beschreiben, daß ein Kunde − aus welchen Gründen auch immer − mehrere verschiedene Kopierer nacheinander benutzen kann, so muß die Für-ein-Verzweigung über *k* zusätzlich in eine Wiederholung eingebettet werden (vgl. Abb. 2.43). Beim Durchlaufen dieses Graphen ist zu beachten, daß man die Für-ein-Verzweigung bei jedem Iterationsschritt *erneut* betritt und daher jedesmal einen *neuen* Wert für *k* wählen kann. Hätte man die Wiederholung *innerhalb* der Für-ein-Verzweigung plaziert (vgl. Abb. 2.44), so würde letztere nur *einmal* durchlaufen, was zur Folge hätte, daß sämtliche Iterationsschritte mit *demselben* Wert für *k* ausgeführt würden. (Faktisch wäre die Wiederholung daher wirkungslos, da ein wiederholtes Benutzen ein und desselben Kopierers bereits durch die Beliebig-oft-Verzweigung innerhalb der Abkürzung Kopierer benutzen erlaubt wird.)

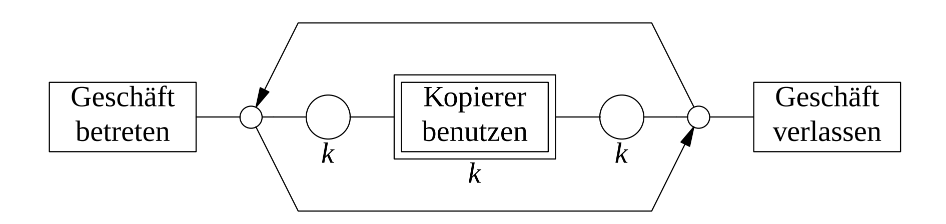


Abbildung 2.43: Wiederholte Für-ein-Verzweigung

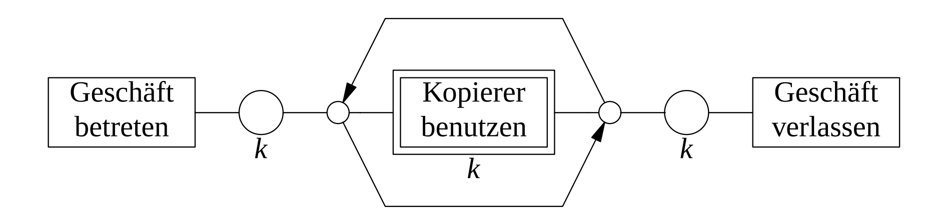


Abbildung 2.44: Fehlerhafte Plazierung der Wiederholung

2.3.5 Zusammenfassung

Tabelle 2.45 faßt die in den zurückliegenden Abschnitten neu eingeführten Operatoren von Interaktionsgraphen noch einmal kurz zusammen und stellt daher − zusammen mit Tab. 2.26 (§ 2.2.6) − eine Kompaktübersicht über die angebotenen Operatoren dar. Außerdem sei an dieser Stelle bereits auf Tab. 2.74 (§ 2.8.1) am Ende des Kapitels verwiesen, in der sowohl die Bezeichnungen als auch die graphischen Darstellungen sämtlicher Operatoren zusammengefaßt sind.

Operator	*Abschnitt*
Beliebig-oft-Verzweigung	2.3.1.3
Kopplung	2.3.2.2
Schablonen	2.3.3
Für-alle-Verzweigung	2.3.4.4
Für-ein-Verzweigung	2.3.4.6

Tabelle 2.45: Weiterführende Operatoren von Interaktionsgraphen

2.4 Zielgerichtetes Durchlaufen von Graphen

Nachdem in den vorangegangenen Abschnitten sämtliche Operatoren von Interaktionsgraphen eingeführt worden sind, sollen in den nun folgenden Abschnitten 2.4 bis 2.6 einige grundlegende konzeptionelle Fragen erörtert werden, bevor in § 2.7 das in Kapitel 1 vorgestellte Anwendungsszenario der interagierenden medizinischen Untersuchungsworkflows wiederaufgegriffen wird.

2.4.1 Willkürliche Entscheidungen

Ein Interaktionsgraph beschreibt eine *Menge zulässiger Ausführungsreihenfolgen*, die man erhält, wenn man den Graphen gemäß der in diesem Kapitel erläuterten Regeln *durchläuft* und die dabei passierten Aktionen *protokolliert*. Wie bereits in § 2.1.1 erwähnt, enthalten die meisten dieser Regeln mehr oder weniger große *Freiheitsgrade*: Beispielsweise kann man an einer Entweder-oder-Verzweigung *entweder* nach oben *oder* nach unten abbiegen, und je nachdem, für welchen Zweig man sich entscheidet, durchläuft man in der Regel unterschiedliche Folgen von Aktionen. An einem Beliebig-oft-Verzweigungsknoten hat man nicht nur zwei, sondern sogar unendlich viele *Alternativen*: Die Mannschaft kann den Rumpf der Verzweigung komplett überspringen oder sich in eine *beliebige* Anzahl von Gruppen aufteilen, die den Rumpf unabhängig voneinander durchlaufen. Ebenso kann z. B. auch der Rumpf einer Wiederholung *beliebig oft* durchlaufen werden.

Fällt man in jeder dieser Entscheidungssituationen eine *willkürliche* oder *zufällige* Entscheidung, d. h. durchläuft man den Graphen *ziellos*, so erhält man als Resultat *irgendeine* zulässige Ausführungsreihenfolge. Die Menge *aller* zulässigen Ausführungsreihenfolgen würde man erhalten, wenn man den Graphen − ähnlich wie ein Labyrinth − systematisch „exploriert“, d. h. *alle* durch die Regeln erlaubten Alternativen verfolgt.

2.4.2 Zielgerichtete Entscheidungen

Beim praktischen Einsatz von Interaktionsgraphen interessiert man sich allerdings weder für die Menge *aller* zulässigen Ausführungsreihenfolgen eines Graphen noch für irgendeine *zufällig* erzeugte Folge von Aktionen. Wesentlich interessanter ist hier die Fragestellung, ob eine *konkret vorliegende* Aktionsfolge aus Sicht des Graphen zulässig ist, und − wenn dies der Fall ist − ob eine bestimmte Aktion im *nächsten Schritt* zulässig ist oder nicht. Insbesondere die zweite Frage, die in § 4.1.1 als *Aktionsproblem* bezeichnet wird, ist für den Einsatz von Interaktionsgraphen als Synchronisationsmechanismus von entscheidender Bedeutung: Wann immer in der realen Welt eine Aktion ausgeführt werden soll, muß anhand des Graphen und der bisher ausgeführten Aktionen entschieden werden können, ob die Aktion zulässig ist oder nicht.

Um diese Frage beantworten zu können, ist es sinnvoll, den gegebenen Graphen nicht willkürlich oder ziellos zu durchlaufen, sondern die tatsächlich in der realen Welt ausgeführten Aktionen als *Richtschnur* oder *Leitlinie* zu verwenden, d. h. erforderliche Entscheidungen *zielgerichtet* zu treffen.

Beispiele

Als einfaches Beispiel betrachte man den Graphen in Abb. 2.46: *Abhängig davon*, ob der Kunde als erstes eine 2-DM- oder eine 1-DM-Münze in den Automaten einwirft, entscheidet man sich beim

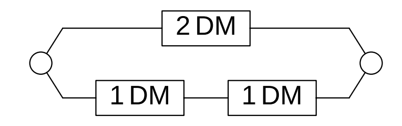


Abbildung 2.46: Einwerfen von 2 DM (vgl. Abb. 2.3, § 2.2.1)

Durchlaufen des Graphen für den oberen bzw. den unteren Zweig der Entweder-oder-Verzweigung. Im ersten Fall (oberer Zweig) erreicht man anschließend sofort das Ende des Graphen, d. h. es sind keine weiteren Aktionen mehr zulässig, während im zweiten Fall (unterer Zweig) das Einwerfen einer weiteren 1-DM-Münze akzeptiert wird.

Ebenso kann man sich beim Durchlaufen des Graphen in Abb. 2.47 in jedem Schritt von den tatsächlich ausgeführten Aktionen leiten lassen:

1. Wie in § 2.2.2.4 erläutert, könnte man im ersten Schritt entweder im oberen Zweig der Sowohl-als-auch-Verzweigung die Aktion *a* durchlaufen oder aber im unteren Zweig *c* passieren.
   Abhängig davon, welche dieser Aktionen in der realen Welt als erstes ausgeführt wird, entscheidet man sich beim Durchlaufen für die entsprechende Alternative. Kommt beispielsweise *c* zur Ausführung, so entscheidet man sich für die Alternative „*c* vor *a*".
2. Im zweiten Schritt könnte man nun entweder *d* oder *a* durchlaufen.
   Wird in der Realität jetzt *a* ausgeführt, so entscheidet man sich für die Alternative „*a* vor *d*".
3. Im dritten Schritt könnte jetzt entweder *b* oder *d* passiert werden.
   Wenn in der Realität *d* ausgeführt wird, entscheidet man sich für die Alternative „*d* vor *b*".
4. Im vierten Schritt kann jetzt nur noch *b* passiert werden, d. h. der Graph akzeptiert nur diese Aktion.
5. Auch im fünften Schritt besteht keine Wahlmöglichkeit mehr; es muß *e* ausgeführt bzw. durchlaufen werden.

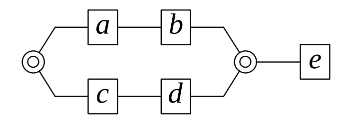


Abbildung 2.47: Sowohl-als-auch-Verzweigung (vgl. Abb. 2.7, § 2.2.2.4)

2.4.3 Aufgeschobene Entscheidungen und deterministisches Verhalten

2.4.3.1 Motivation

Bei vielen Graphen erlaubt die Ausführung der jeweils nächsten Aktion allerdings noch keine endgültige Entscheidung, welche der prinzipiell möglichen Alternativen zu wählen ist. Als einfaches Beispiel betrachte man den Graphen in Abb. 2.48, an dessen Entweder-oder-Verzweigungsknoten man prinzipiell drei Alternativen zur Auswahl hat. Wird als erstes eine 2-DM-Münze eingeworfen, so ist der Fall klar: man durchläuft den unteren Zweig und akzeptiert im nächsten Schritt nur noch eine 1-DM-Münze. Beim Einwurf einer 1-DM-Münze im ersten Schritt entsteht jedoch eine *Konfliktsitua-*

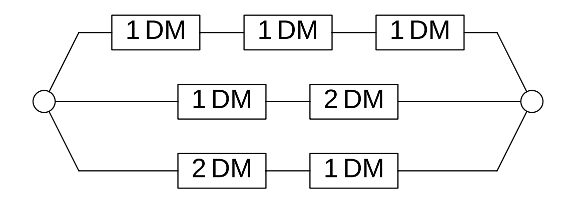


Abbildung 2.48: Einwerfen von 3 DM (vgl. Abb. 2.4, § 2.2.2.1)

tion, da zu diesem Zeitpunkt noch nicht entschieden werden kann, ob der obere oder der mittlere Zweig gewählt werden soll: Wenn der Kunde beabsichtigt, zwei weitere 1-DM-Münzen einzuwerfen, sollte man den oberen Zweig durchlaufen; will er anschließend jedoch eine 2-DM-Münze einwerfen, müßte man sich für den mittleren Zweig entscheiden.

Da der Graph (bzw. ein Programm, das ihn interpretiert) die Absicht des Kunden nicht kennt − und der Kunde auch keine Möglichkeit hat, diese zu kommunizieren −, gibt es prinzipiell nur zwei mögliche Verhaltensweisen:

1. Der Graph verhält sich *nichtdeterministisch*, d. h. er entscheidet sich − ähnlich wie in § 2.4.1 beschrieben − willkürlich oder zufällig für eine der beiden Alternativen.
2. Der Graph verhält sich *deterministisch*, indem er die Entscheidung *aufschiebt* und *beide* Alternativen so lange weiterverfolgt, bis sich eine von ihnen definitiv als falsch erweist, d. h. mit der tatsächlich ausgeführten Folge von Aktionen nicht mehr vereinbar ist.

Obwohl die erste Variante aus Entwicklersicht den Vorteil besitzt, daß sie sehr leicht implementiert werden kann, ist sie aus Anwendersicht jedoch kaum akzeptabel, da die Benutzung eines Automaten so zum Glücksspiel wird: Wirft der Kunde im ersten Schritt eine 1-DM-Münze ein, so hängt es vom Zufall ab, ob anschließend eine 2-DM- oder aber zwei 1-DM-Münzen akzeptiert werden; hat der Kunde gerade nur die andere Alternative zur Hand, so hat er „Pech gehabt“. Da ein solches Systemverhalten in den meisten Anwendungen nicht zumutbar ist (der Kunde bzw. Anwender soll schließlich „König“ sein und nicht der Willkür des Systems ausgeliefert sein), muß sich der Automat, d. h. der Interaktionsgraph bzw. das ihn interpretierende Programm, gemäß Variante 2 verhalten, d. h. Entscheidungen, die momentan nicht getroffen werden können, solange aufschieben, bis sie getroffen werden können.

Im konkreten Beispiel bedeutet das, daß man nach Einwurf einer 1-DM-Münze im ersten Schritt *sowohl* die Alternative „oberer Zweig“ *als auch* die Alternative „mittlerer Zweig“ weiterverfolgen muß. Wird als nächstes eine weitere 1-DM-Münze eingeworfen, so wird „rückwirkend“ klar, daß man den oberen Zweig durchlaufen muß, da der mittlere Zweig diese Aktion im zweiten Schritt nicht erlaubt. Daher muß die Alternative „mittlerer Zweig“ jetzt verworfen und nur noch die Alternative „oberer Zweig“ weiterverfolgt werden. Wird im zweiten Schritt jedoch eine 2-DM-Münze eingeworfen, so verhält es sich gerade umgekehrt: Die Alternative „oberer Zweig“ wird verworfen, weil sie die Aktion 2 DM im zweiten Schritt nicht erlaubt, während die Alternative „mittlerer Zweig“ weiterverfolgt wird.

2.4.3.2 Einwand

Man mag an dieser Stelle einwenden, daß man den Graphen in Abb. 2.48 unmittelbar in einen äquivalenten Graphen transformieren kann, bei dem die beschriebenen Entscheidungsprobleme oder Konflikte nicht auftreten, indem man die „kritische“ Aktion 1 DM aus den oberen beiden Zweigen der Entweder-oder-Verzweigung ausklammert (vgl. Abb. 2.49). Allerdings ist ein Graphautor möglicherweise weder imstande, derartige Konflikte auf Anhieb zu erkennen, noch willens, sie durch Äquivalenztransformationen zu eliminieren. Frei nach dem Motto: „Lassen Sie andere Leute bzw. die Maschine die Arbeit tun“ [Kernighan86], sollte ein Graphautor daher größtmögliche Freiheit bei der Formulierung von Graphen genießen und nicht durch derartige, im Grunde genommen implementierungstechnische Probleme belastet werden.

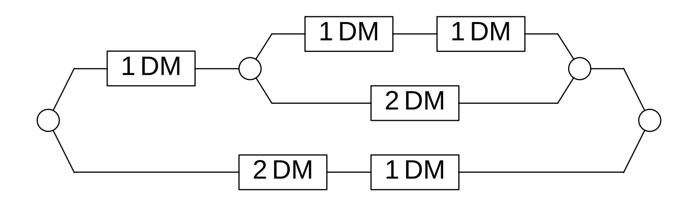


Abbildung 2.49: Einwerfen von 3 DM ohne Konflikte

2.4.3.3 Weitere Beispiele

Andere Graphen, wie z. B. die in Abb. 2.16 bis 2.18 (§ 2.2.4.3), enthalten Konflikte, die nicht ohne weiteres durch Äquivalenztransformationen eliminiert werden können. Beispielsweise enthalten alle diese Graphen die Aktion 1 DM *formal* zweimal und *real* (wenn man die Mehrfach-Verzweigungen expandiert) sogar fünfmal. Aufgrund der Struktur der Graphen könnte jede dieser fünf Ausprägungen im ersten Schritt durchlaufen werden, d. h. wenn als erstes eine 1-DM-Münze in den Automaten eingeworfen wird, muß der Graph-Interpreter anschließend fünf verschiedene Alternativen weiterverfolgen.

Wollte man diese Konflikte eliminieren, so könnte man zunächst alle Mehrfach- bzw. Sowohl-als-auch-Verzweigungen in äquivalente Entweder-oder-Verzweigungen transformieren, was prinzipiell möglich ist, da alle Zweige *endlich* sind, d. h. weder Wiederholungen noch Beliebig-oft-Verzweigungen enthalten. Das Resultat dieser Transformationen entspricht einer „primitiven“ Aufzählung aller möglichen Münzfolgen wie in Abb. 2.15, das man durch Ausklammern gemeinsamer Präfixe (ähnlich wie oben) konfliktfrei machen könnte (vgl. Abb. 2.50). Würde man dieses Verfahren jedoch auf die Graphen in Abb. 2.17 und 2.18 anwenden, so ergäben sich Graphen, deren Größe − wie bereits in § 2.2.4.3 erläutert − jedes vernünftige und praktisch handhabbare Maß übersteigen würde.

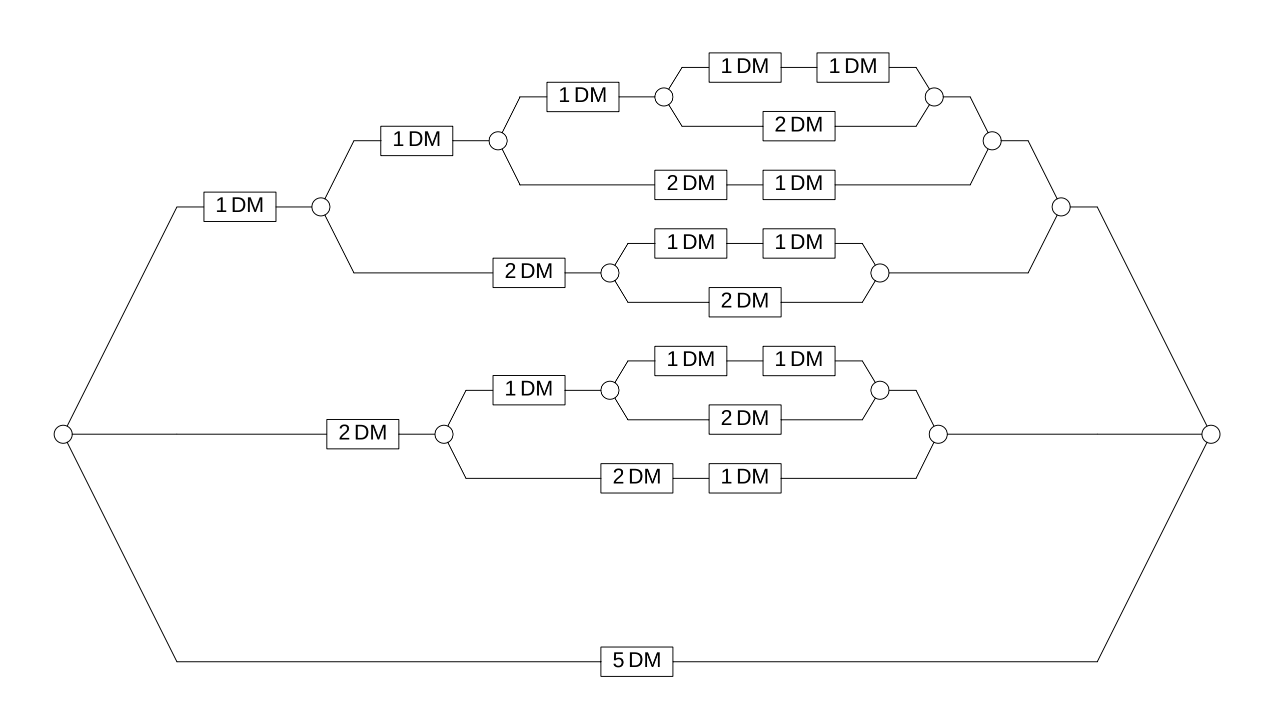


Abbildung 2.50: Einwerfen von 5 DM ohne Konflikte

2.4.3.4 Inhärent konfliktbehaftete Graphen

Schließlich gibt es auch Graphen, wie z. B. die Beschreibung des Komfort-Münzkopierers in Abb. 2.51, die *inhärent konfliktbehaftet* sind, d. h. deren Konflikte sich grundsätzlich *nicht* durch Äquivalenztransformationen eliminieren lassen. Dies liegt darin begründet, daß eine Beliebig-oft-Verzweigung im Prinzip eine *unendliche* Entweder-oder-Verzweigung wie in Abb. 2.52 darstellt, für die das oben beschriebene Transformationsverfahren nicht terminieren würde.

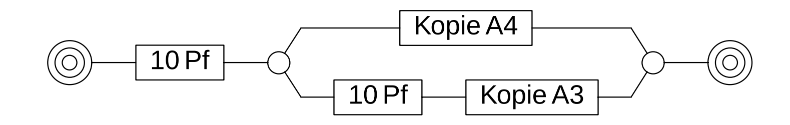


Abbildung 2.51: Komfort-Münzkopierer (vgl. Abb. 2.30, § 2.3.1.4)

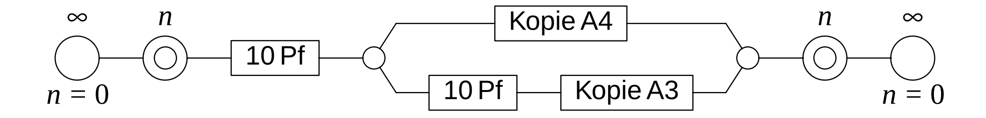


Abbildung 2.52: Beliebig-oft-Verzweigung als unendliche Entweder-oder-Verzweigung

2.4.3.5 Resümee

Aus all diesen Gründen können und dürfen Interaktionsgraphen Konflikte enthalten, die beim zielgerichteten Durchlaufen dazu führen, daß das interpretierende Programm mehrere Alternativen parallel verfolgen muß, um ein deterministisches Systemverhalten zu gewährleisten, das die *Souveränität des Benutzers* − d. h. sein Recht, jederzeit eine *beliebige* momentan zulässige Aktion auszuführen − respektiert.

2.4.4 Komplexitätsbetrachtungen

2.4.4.1 Exponentielle Komplexität

Versucht man den Graphen in Abb. 2.51 nach diesem Prinzip zu durchlaufen, so stellt man allerdings fest, daß die Anzahl der parallel zu verfolgenden Alternativen sehr groß werden kann:

0. Zunächst stellt sich die Frage, in wieviele Gruppen man die Gesamtmannschaft am linken ◎-Knoten aufteilen soll. Die Beantwortung dieser Frage läßt sich allerdings dadurch aufschieben, daß man nicht a priori eine bestimmte Anzahl von Gruppen festlegt, sondern Gruppen nach und nach „abspaltet“. Man beginnt also mit einer einzelnen Gruppe G_1, die anfängt, den Rumpf der Verzweigung zu durchlaufen.

1. Wird in diesem initialen Zustand die einzig zulässige Aktion 10 Pf ausgeführt, so muß die Gruppe G_1 die linke Aktion 10 Pf durchlaufen.

2. Wird die Aktion 10 Pf ein zweites Mal ausgeführt, gibt es bereits zwei Alternativen:

   a) Entweder setzt die Gruppe G_1 ihre Reise fort, biegt am Entweder-oder-Verzweigungsknoten nach unten ab und durchläuft die rechte Aktion 10 Pf,

   b) oder man spaltet am linken ◎-Knoten eine zweite Gruppe G_2 ab, die die linke Aktion 10 Pf durchläuft.

   Formal kann man die resultierenden *Zustände* wie folgt mit Hilfe von Tupeln beschreiben:

   a) Das *Ein*-Tupel $[2]$ besagt, daß es *eine* Gruppe G_1 gibt, die 2-mal die Aktion 10 Pf durchlaufen hat,

   b) während das *Zwei*-Tupel $[1, 1]$ anzeigt, daß es *zwei* Gruppen G_1 und G_2 gibt, die beide 1-mal die Aktion 10 Pf passiert haben.

   Allgemein repräsentiert ein n-Tupel $[k_1, \ldots, k_n]$ mit $k_1, \ldots, k_n \in \{ 1, 2 \}$ einen Zustand, in dem

n Gruppen $G_1, \ldots, G_n$ im Rumpf der Beliebig-oft-Verzweigung unterwegs sind und jede Gruppe G_i die Aktion 10 Pf k_i-mal durchlaufen hat ($i = 1, \ldots, n$). (Die Gesamtzahl der ausgeführten 10 Pf-Aktionen entspricht daher der Quersumme $k_1 + \ldots + k_n$ des Tupels.)

3. Wird die Aktion 10 Pf ein drittes Mal ausgeführt, ergeben sich folgende drei Alternativen:
   a) Ausgehend vom Zustand $[1, 1]$, wird eine dritte Gruppe G_3 abgespalten, die die linke Aktion 10 Pf durchläuft.
   Resultierender Zustand: $[1, 1, 1]$.
   b) Ausgehend vom selben Zustand $[1, 1]$, geht eine der beiden Gruppen weiter und durchläuft die rechte Aktion 10 Pf.
   Resultat: $[1, 2]$ oder $[2, 1]$.
   c) Ausgehend vom Zustand $[2]$, wird eine zweite Gruppe G_2 abgespalten, die die linke Aktion 10 Pf durchläuft.
   Resultat ebenfalls: $[2, 1]$.
4. Wird die Aktion 10 Pf ein viertes Mal ausgeführt, ergeben sich die folgenden fünf Alternativen:
   a) Ausgehend vom Zustand $[1, 1, 1]$, wird eine vierte Gruppe G_4 abgespalten, die die linke Aktion 10 Pf durchläuft.
   Resultat: $[1, 1, 1, 1]$.
   b) Ausgehend vom selben Zustand $[1, 1, 1]$, geht eine der drei Gruppen weiter und durchläuft die rechte Aktion 10 Pf.
   Resultat: $[1, 1, 2]$ oder $[1, 2, 1]$ oder $[2, 1, 1]$.
   c) Ausgehend vom Zustand $[1, 2]$, wird eine dritte Gruppe G_3 abgespalten, die die linke Aktion 10 Pf durchläuft.
   Resultat erneut: $[1, 2, 1]$.
   d) Ausgehend vom selben Zustand $[1, 2]$, geht die Gruppe G_1 weiter und durchläuft die rechte Aktion 10 Pf.
   Resultat: $[2, 2]$.
   e) Ausgehend vom symmetrischen Zustand $[2, 1]$, erhält man entsprechend einen der Zustände $[2, 1, 1]$ oder wiederum $[2, 2]$.
5. Usw.

Wie die folgenden Überlegungen zeigen, erhält man auf diese Weise nach $2n$ Ausführungen der Aktion 10 Pf eine Anzahl N von mehr als 2^n verschiedenen Tupeln (die alle die Quersumme $2n$ besitzen, siehe oben), d. h. die Anzahl der zu verfolgenden Alternativen wächst in diesem Beispiel (mindestens) *exponentiell* bzgl. der Länge der Eingabefolge:

- Es gibt genau ein Tupel, das aus $2n$ Einsen besteht.
- Es gibt $(2n{-}1)$ verschiedene Tupel, die aus $2n{-}2$ Einsen und einer Zwei bestehen. (Man hat $2n{-}1$ Möglichkeiten, die eine Zwei im Tupel zu plazieren.)
- Es gibt $\binom{2n-2}{2}$ verschiedene Tupel, die aus $2n{-}4$ Einsen und zwei Zweien bestehen.
- Es gibt $\binom{2n-3}{3}$ verschiedene Tupel, die aus $2n{-}6$ Einsen und drei Zweien bestehen.
- Usw.
- Schließlich gibt es $\binom{2n-n}{n} = 1$ Tupel, das aus n Zweien besteht.

Für die Gesamtzahl N der verschiedenen Tupel gilt daher:

$$N = \binom{2n}{0} + \binom{2n-1}{1} + \binom{2n-2}{2} + \binom{2n-3}{3} + \ldots + \binom{2n-n}{n} = \sum_{k=0}^{n} \binom{2n-k}{k} > \sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

2.4.4.2 Lineare Komplexität

Würde man diese Beobachtung so stehenlassen, müßte man Interaktionsgraphen als Formalismus bezeichnen, der zwar konzeptionell interessant und ausdrucksstark, praktisch aber aufgrund seiner potentiellen Komplexität nicht einsetzbar ist. Schließlich handelt es sich bei dem betrachteten Graphen in Abb. 2.30 durchaus um ein realistisches und rein syntaktisch auch sehr einfaches Beispiel.

Glücklicherweise ist die Situation jedoch bei weitem nicht so dramatisch, wie sie auf den ersten Blick erscheinen mag. Betrachtet man die resultierenden Zustände bzw. Alternativen nämlich etwas genauer, so stellt man fest, daß viele von ihnen zwar formal verschieden, im Prinzip aber doch *äquivalent* sind. Beispielsweise spielt es keine Rolle, *welche* der zwei Gruppen G_1 und G_2 im Zustand [1, 1] weitergeht und die rechte Aktion 10 Pf durchläuft, d. h. die Zustände [1, 2] und [2, 1] sind in gewisser Weise gleichwertig. Ebenso sind die Zustände [1, 1, 2], [1, 2, 1] und [2, 1, 1] äquivalent, weil sie alle drei besagen, daß die Aktion 10 Pf von zwei Gruppen je einmal und von einer Gruppe zweimal durchlaufen wurde; *welche* der drei Gruppen die Aktion zweimal durchlaufen hat, ist wiederum unerheblich.

Formal bedeutet das, daß zwei Zustandstupel äquivalent sind, wenn sie durch eine geeignete *Permutation* ineinander überführt werden können. Aus diesem Grund ist es sinnvoller, Zustände nicht wie bisher durch geordnete Tupel, sondern durch ungeordnete *Multimengen* zu beschreiben, die zwar Elemente mehrfach enthalten können, bei denen die *Reihenfolge* der Elemente jedoch *irrelevant* ist. Für die Berechnung der Anzahl N der zu verfolgenden Alternativen bedeutet das:

- Es gibt genau *eine* Multimenge, die $2n$ Einsen enthält.
- Es gibt genau *eine* Multimenge, die $2n-2$ Einsen und eine Zwei enthält.
- Es gibt genau *eine* Multimenge, die $2n-4$ Einsen und zwei Zweien enthält.
- Es gibt genau *eine* Multimenge, die $2n-6$ Einsen und drei Zweien enthält.
- Usw.
- Es gibt genau *eine* Multimenge, die n Zweien enthält.

Somit gibt es nach $2n$ Ausführungen der Aktion 10 Pf nur noch $n+1$ verschiedene Zustände bzw. Alternativen, d. h. die Komplexität konnte von exponentiell auf linear reduziert werden!

2.4.5 Zusammenfassung

In den vorangegangenen Überlegungen wurden im Prinzip schon einige Kerngedanken einer praktischen *Implementierung* von Interaktionsgraphen vorweggenommen (vgl. Kapitel 4), die im folgenden noch einmal kurz zusammengefaßt werden:

1. Um entscheiden zu können, ob eine bestimmte Aktion in einer bestimmten Situation zulässig ist oder nicht, ist es sinnvoll, den gegebenen Graphen *zielgerichtet* zu durchlaufen, d. h. die tatsächlich ausgeführten Aktionen in notwendige Entscheidungsprozesse miteinzubeziehen.

2. Aus verschiedenen Gründen können und dürfen Graphen *Konflikte* enthalten, d. h. beim zielgerichteten Durchlaufen können Situationen auftreten, in denen Entscheidungen nicht *sofort* getroffen werden können.

3. Um ein *deterministisches Systemverhalten* zu gewährleisten, das die *Souveränität des Benutzers* respektiert, dürfen in solchen Situationen keine willkürlichen oder zufälligen Entscheidungen getroffen werden, weil diese im Widerspruch zur Intention des Benutzers stehen könnten. Vielmehr

müssen Entscheidungen solange *aufgeschoben* werden, bis sie *deterministisch* getroffen werden können.

4. Bei einer naiven Umsetzung dieses Prinzips kann die Menge der zu verwaltenden Zustände oder Alternativen Größenordnungen annehmen, die praktisch nicht mehr handhabbar sind. Daher besteht ein wesentliches Ziel bei der Entwicklung einer *effizienten* Implementierung darin, Zustände so zu definieren und zu verwalten, daß möglichst viele *äquivalente* Zustände als solche erkannt werden können und so die Menge der parallel zu verfolgenden Alternativen möglichst klein bleibt. Wie in § 4.7.5.4 erläutert wird, konnte dieses Ziel für eine sehr große Klasse praktisch relevanter Graphen tatsächlich erreicht werden.

2.5 Sackgassen und endlose Wege

2.5.1 Sackgassen

2.5.1.1 Beispiel

Versucht man, den Graphen in Abb. 2.53 nach den vorgeschriebenen Regeln zu traversieren, so ergibt sich eine merkwürdige Situation: Nach Passieren der Aktion *a* müssen die Zweige der Kopplung parallel durchlaufen werden. Da die Aktionen *b* und *c* in *beiden* Zweigen auftreten, müssen sie auch in beiden Zweigen *gleichzeitig* durchlaufen werden; dies ist aber offensichtlich nicht möglich, da im oberen Zweig zuerst *b* und anschließend *c* passiert werden muß, während die Reihenfolge im unteren Zweig gerade umgekehrt ist. Somit stellt die Kopplung o···o eine *Sackgasse* dar, in der man beim Durchlaufen des Graphen „steckenbleibt."

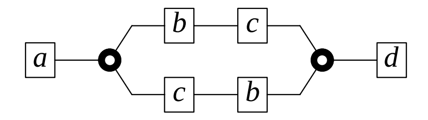


Abbildung 2.53: Graph mit einer Sackgasse

2.5.1.2 Diskussion

Obwohl die Formulierung von Sackgassen in aller Regel nicht sinnvoll ist, kann man sie rein *syntaktisch* nicht unterbinden, sofern man auf die für praktische Anwendungen sehr wichtige *Kopplung* von Graphen nicht verzichten will.

Prinzipiell könnte man nun versuchen, *semantische* Kriterien zu finden, mit deren Hilfe für einen gegebenen Graphen überprüft werden kann, ob er *sackgassenfrei* ist (d. h. ob es mindestens einen gangbaren Weg durch den Graphen gibt) oder nicht. Aus verschiedenen Gründen wurde diese Frage im Rahmen dieser Arbeit jedoch nicht weiter verfolgt:

1. Zuallererst müßte geklärt werden, ob das Sackgassen- bzw. *Erfüllbarkeitsproblem*[7] für Interaktionsgraphen (bzw. -ausdrücke) überhaupt *entscheidbar* ist. Da die Ausdrucksmächtigkeit von Interaktionsgraphen teilweise über die von kontextfreien Grammatiken hinausgeht (vgl. § 3.5.3) und das Erfüllbarkeitsproblem für kontextsensitive Grammatiken bekanntlich *nicht* entscheidbar ist [Hopcroft90, Schöning95], ist die Antwort auf diese Frage keineswegs offensichtlich und ihre Klärung möglicherweise schwierig und aufwendig.

[7] Gemeint ist die Frage, ob die durch einen Graphen bzw. Ausdruck definierte *Sprache* (vgl. § 3.2.3) leer ist oder nicht. In der Literatur wird dies auch als Leerheits- oder Leere-Problem bezeichnet [Schöning95, Hopcroft90].

2. Sofern das Problem entscheidbar ist, müßte als nächstes untersucht werden, ob es auch *effizient* entscheidbar ist. Wenn auch dies der Fall ist, müssen konkrete Kriterien formuliert werden, die für beliebige Graphen entscheiden, ob sie sackgassenfrei sind oder nicht. Auch die Formulierung und Verifizierung derartiger Kriterien dürfte mit erheblichem Aufwand verbunden sein.

3. Schließlich stellen Sackgassen sicherlich nicht die einzige Möglichkeit dar, mit Interaktionsgraphen „semantischen Unsinn“ zu formulieren, den man häufig nur mit entsprechendem *Anwendungswissen* als solchen erkennen kann. Beispielsweise ist der Graph in Abb. 2.54 aus Anwendungssicht ziemlich unsinnig (nach einmaligem Einwurf von 5 DM werden beliebig viele Zigarettenpackungen ausgegeben), obwohl er rein formal vollkommen in Ordnung ist. Aus diesem Grund ist es nicht sehr rentabel, für die Erkennung einer speziellen Teilklasse unsinniger Ausdrücke einen hohen formalen Aufwand zu treiben, während eine wesentlich größere Klasse derartiger Ausdrücke grundsätzlich nicht als solche erkannt werden kann.

Dies bedeutet, daß die Verantwortung zur Formulierung *sinnvoller* Interaktionsgraphen bewußt dem jeweiligen Graphautor überlassen bleibt, ebenso wie z. B. ein Programmierer oder ein Workflow-Modellierer dafür verantwortlich ist, die ihm zur Verfügung stehende Programmier- oder Workflowbeschreibungs-Sprache sinnvoll einzusetzen.

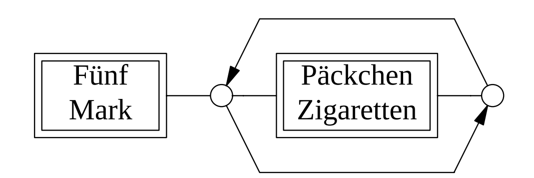


Abbildung 2.54: Semantisch unsinniger Graph

Bei der Entwicklung der formalen Semantik muß das Problem der Sackgassen jedoch − ebenso wie das im folgenden vorgestellte Phänomen der endlosen Wege − in geeigneter Weise berücksichtigt werden (vgl. § 3.2.3).

2.5.2 Endlose Wege

2.5.2.1 Beispiele

Auch der Graph in Abb. 2.55 zeigt ein merkwürdiges Verhalten, wenn man ihn gemäß der Regeln aus § 2.3.4.4 durchläuft: Aufgrund der dortigen Anmerkung teilt sich die Mannschaft am ◎-Verzweigungsknoten in *unendlich* viele Gruppen auf, die unabhängig voneinander den Rumpf der Für-alle-Verzweigung durchlaufen. Da jede dieser Gruppen die Aktion a durchlaufen muß, um den ◎-Vereinigungsknoten zu erreichen, müßten insgesamt *unendlich* viele Aktionen a passiert werden, bevor die Mannschaft an diesem Knoten wieder komplett vereinigt ist. Der Graph enthält somit unendlich lange bzw. *endlose Wege*.

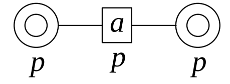


Abbildung 2.55: Graph mit endlosen Wegen

Da in einem endlichen Betrachtungszeitraum aber offensichtlich nur endlich viele Aktionen ausgeführt werden können (vgl. auch § 2.6.1), kann ein solcher Graph nicht in endlicher Zeit durchlaufen werden. Dies hat zur Folge, daß eventuell nachfolgende Aktionen (wie z. B. die Aktion *b* in Abb. 2.56) niemals erreicht werden, d. h. niemals zulässig sind. Somit akzeptieren die beiden Graphen in Abb. 2.55 und 2.56 exakt *dieselben* (endlichen!) Aktionsfolgen und können daher als *fast äquivalent* bezeichnet werden.[8]

Abbildung 2.56: Fast äquivalenter Graph mit endlosen Wegen

Anders verhält es sich mit dem Graphen in Abb. 2.57: Obwohl sich die Mannschaft auch hier am ◎-Verzweigungsknoten in *unendlich* viele Gruppen aufteilt, kann sie ihre Reise *jederzeit* am ◎-Vereinigungsknoten *gemeinsam* fortsetzen, weil jede der Gruppen die Möglichkeit hat, diesen Knoten direkt, d. h. *ohne Passieren einer Aktion*, zu erreichen.

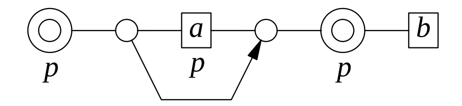


Abbildung 2.57: Graph ohne endlose Wege

2.5.2.2 Folgerung

Aus der Betrachtung dieser einfachen Beispiele folgt anschaulich sofort die folgende Behauptung, die in § 3.4.6 auch formal bewiesen wird: Eine Für-alle-Verzweigung enthält genau dann endlose Wege, wenn ihr Rumpf *keinen leeren Weg* enthält. Interessant ist in diesem Zusammenhang auch, daß die Frage, ob ein Graph einen leeren Weg enthält, sehr einfach *syntaktisch* entschieden werden kann. Auch dies wird in § 3.4.6 gezeigt.

2.6 Aktionen und Aktivitäten

2.6.1 Punktuelle Aktionen

Bereits in § 2.2.2.3 wurde erwähnt, daß zwei Münzen immer in einer bestimmten *Reihenfolge* in einen Automaten eingeworfen werden müssen, weil er normalerweise nur einen Münzschlitz besitzt. Das bedeutet, daß sich zwei Ausführungen der Tätigkeit „Münze einwerfen“ zeitlich *nicht überlappen* können. Da es außerdem schwierig ist, die Ausführung einer solchen Tätigkeit zeitlich exakt zu fassen

[8] In § 3.4.1.1 werden zwei Graphen als *äquivalent* definiert, wenn sie dieselben Aktionsfolgen akzeptieren *und* dieselben Aktionen enthalten. Da letzteres für die beiden Graphen nicht zutrifft, sind sie nur *fast* äquivalent. Würde man sie beispielsweise jeweils mit einer Aktion *b* koppeln, so könnte diese im einen Fall (Abb. 2.55 gekoppelt mit *b*) jederzeit durchlaufen werden (da sie nur im Zweig *b* auftritt), während sie im anderen Fall (Abb. 2.56 gekoppelt mit *b*) niemals passiert werden könnte (da sie in beiden Zweigen der Kopplung auftritt, aber nur im Zweig *b* durchlaufen werden könnte). Somit besitzen die beiden Graphen an sich zwar dasselbe Verhalten, als Teilgraphen eines größeren Graphen können sie jedoch unterschiedlichen Einfluß auf *dessen* Verhalten haben.

(wann genau beginnt bzw. endet das Einwerfen einer Münze?), kann man darüber hinaus vereinfachend bzw. idealisierend annehmen, daß die Ausführung einer solchen Tätigkeit *keine zeitliche Ausdehnung* besitzt, d. h. ein *punktuelles Ereignis* auf der Zeitachse darstellt. Schließlich ist es sinnvoll anzunehmen, daß in jedem endlichen Zeitintervall nur *endlich viele* Münzen in einen Automaten eingeworfen werden können.

Eine Tätigkeit, die diesen Kriterien entspricht, wird im folgenden als *Aktion* bezeichnet. Allgemein können Aktionen daher wie folgt charakterisiert werden:

1. Aktionen besitzen *keine zeitliche Ausdehnung*, d. h. ihre Ausführung stellt ein *punktuelles Ereignis* auf der Zeitachse dar.
2. Aktionen können *nicht gleichzeitig* ausgeführt werden.
3. In einem endlichen Betrachtungszeitraum können nur *endlich viele* Aktionen ausgeführt werden.

2.6.2 Zeitlich ausgedehnte und überlappende Aktivitäten

In Anwendungsgebieten wie z. B. Workflow-Management trifft man jedoch häufig auf Tätigkeiten, für die die ersten beiden Kriterien nicht zutreffen, d. h. die sowohl *zeitlich ausgedehnt* sind als auch *zeitlich überlappend* ausgeführt werden können. Beispielsweise kann sich die Ausführung eines Workflowschritts mit Hilfe eines Schrittprogramms durchaus über mehrere Minuten, Stunden oder sogar Tage erstrecken (wenn beispielsweise mit Hilfe eines Textverarbeitungsprogramms ein größeres Dokument zu erstellen ist), und selbstverständlich können mehrere derartige Schritte (z. B. aus verschiedenen Workflows) auch zeitlich überlappend ausgeführt werden. Derartige Tätigkeiten, die sowohl zeitlich ausgedehnt sind als auch zeitlich überlappend ausgeführt werden können, werden im folgenden als *Aktivitäten* bezeichnet.

Da die tatsächliche Dauer einer Aktivität A jedoch von untergeordneter Bedeutung ist, kann ihre Ausführung durch eine Folge $A_0 - A_1$ zweier punktueller Aktionen A_0 und A_1 beschrieben werden, die das *Starten* bzw. *Beenden* der Aktivität A bezeichnen und zwischen deren Ausführung beliebig viel Zeit verstreichen kann. Auch eine zeitlich überlappende Ausführung mehrerer Aktivitäten kann so auf eine *sequentielle* Ausführung *punktueller* Aktionen zurückgeführt werden (vgl. Abb. 2.58), sofern man bereit ist zu akzeptieren, daß eine Aktivität nicht exakt im selben Moment gestartet oder beendet werden kann, in dem eine andere Aktivität gestartet oder beendet wird.

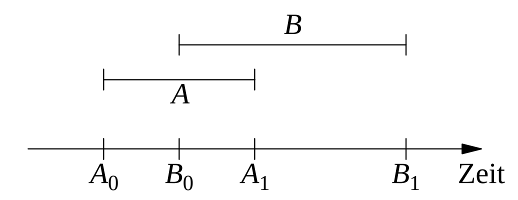


Abbildung 2.58: Zeitlich überlappende Ausführung von Aktivitäten

Vergleicht man die Ausführung von Aktivitäten mit der von Programmen auf einem Einprozessorrechner, so werden zwei Programme vom Betriebssystem tatsächlich immer in einer bestimmten Reihenfolge gestartet, selbst wenn die beiden Benutzer, die ihre Ausführung veranlassen, gleichzeitig „die Returntaste drücken". Auf einem Mehrprozessorsystem oder in einem Rechnernetz wäre es zwar prinzipiell möglich, daß zwei Programme exakt gleichzeitig gestartet (oder beendet) werden; sofern man jedoch sämtliche Start- und Endeaktionen in einer *gemeinsamen* Protokolldatei aufzeichnet, erhält man auch hier wieder eine rein sequentielle Folge von Aktionen, in der gleichzeitig ausgeführte Aktionen durch geeignete Synchronisationsmechanismen bei der Protokollschreibung „künstlich" in eine bestimmte Reihenfolge gebracht werden.

Beim konkreten Einsatz von Interaktionsgraphen zur Synchronisation von Workflows oder sonstigen Prozessen ist das Szenario sehr ähnlich: Die zu synchronisierenden Workflowschritte oder Aktivi-

täten können zwar prinzipiell von verschiedenen Benutzern an verschiedenen Rechnern gestartet werden, ihre Ausführung muß jedoch von einem *zentralen Interaktionsmanager*[9] verfolgt und kontrolliert werden, der dafür sorgt, daß zu jedem Zeitpunkt nur zulässige Aktionen ausgeführt werden (vgl. § 5.2.1 und § 5.5). Somit werden auch hier Aktionen immer in einer bestimmten Reihenfolge ausgeführt.

2.6.3 Anmerkungen

In der Literatur wird für Tätigkeiten ohne zeitliche Ausdehnung und Überlappung gelegentlich auch der Begriff *Ereignis* (engl. event) verwendet [Klein91, Attie93, Tang95]. Aus sprachlichen Gründen wird in dieser Arbeit jedoch die Bezeichnung *Aktion* (engl. action) bevorzugt: Ereignisse *ereignen* sich, d. h. sie können von Natur aus weder ausgeführt werden noch können sie zulässig oder unzulässig sein − sie treten einfach ein. Im Gegensatz hierzu können und müssen Aktionen explizit ausgeführt werden (indem jemand in *Aktion* tritt), und diese Ausführung kann zulässig sein oder nicht.

Der Begriff *Aktivität* (engl. activity) für eine zeitlich ausgedehnte Tätigkeit ist hingegen allgemein üblich und wird z. B. auch von der Workflow Management Coalition (WfMC) als Bezeichnung für die elementaren Arbeitsschritte eines Workflows verwendet [WfMC96].

Die formale Reduktion zeitlich ausgedehnter und überlappender Aktivitäten auf punktuelle, sequentiell ausgeführte Aktionen ist eine gebräuchliche Methode, die es erlaubt, das Verhalten nebenläufiger Systeme mit Hilfe *formaler Sprachen*, *Spuren* (engl. traces) o. ä. zu beschreiben [Guo96, Shaw78, Hoare85, Harel87] (vgl. auch § 3.3). Die zusätzliche Vereinbarung, daß in jedem endlichen Betrachtungszeitraum nur endliche viele Aktionen ausgeführt werden können, erlaubt es, die Betrachtung auf *endliche Aktionsfolgen* zu beschränken.

2.6.4 Graphische Darstellung

In der graphischen Darstellung werden punktuelle Aktionen wie bisher durch einfache Rechtecke dargestellt. Zeitlich ausgedehnte Aktivitäten, wie z. B. eine Aktivität schlafen, müßten prinzipiell als Sequenz schlafen_0 – schlafen_1 dargestellt werden (Abb. 2.59). Um jedoch anzudeuten, daß eine solche Sequenz eine logische Einheit darstellt, werden die beiden Aktionen zu einem einzigen Rechteck zusammengefaßt, das durch einen senkrechten Strich in zwei Hälften unterteilt wird, die die Start- bzw. Endeaktion der Aktivität repräsentieren (Abb. 2.60). Konzeptuell kann ein solches unterteiltes Rechteck als implizit definierte Abkürzung der entsprechenden Sequenz aufgefaßt werden.

schlafen_0 — schlafen_1

Abbildung 2.59: Aktivität als Sequenz zweier Aktionen

schl|afen

Abbildung 2.60: Aktivität als logische Einheit

[9] Ein Interaktionsmanager ist zumindest für einen bestimmten Graphen zentral, d. h. Aktionen, über die dieser Graph eine Aussage macht, können nur mit Zustimmung dieses Managers ausgeführt werden. Verschiedene Graphen können aber durchaus von verschiedenen Interaktionsmanagern verwaltet werden, um Engpässe bei der Verarbeitung von Aktionen zu vermeiden (vgl. § 5.2.7.1).

2.7 Interagierende medizinische Untersuchungsworkflows

Nachdem die verschiedenen Operatoren und Prinzipien von Interaktionsgraphen in den vorangegangenen Abschnitten dieses Kapitels ausführlich erläutert und diskutiert worden sind, ist man nunmehr in der Lage, das in Kapitel 1 vorgestellte Anwendungsszenario der interagierenden medizinischen Untersuchungsworkflows wiederaufzugreifen und konkrete Lösungsvorschläge für die aufgeworfenen Probleme zu entwickeln.

2.7.1 Integritätsbedingungen für Patienten

2.7.1.1 Bedingung für einen Patienten

Bereits in § 1.2.2 wurde erwähnt, daß sich ein Patient nicht in zwei Untersuchungsstellen gleichzeitig befinden kann, d. h. daß die Schrittfolge Patient abrufen − Untersuchung durchführen[10] einen *kritischen Abschnitt* [BrinchHansen72] darstellt, der von parallel ausgeführten Untersuchungsworkflows dieses Patienten nicht gleichzeitig durchlaufen werden darf. Außerdem kann ein Patient, während er sich in einer Untersuchungsstelle befindet, nicht für eine andere Untersuchung vorbereitet oder aufgeklärt werden, d. h. die zugehörigen Tätigkeiten schließen sich wechselseitig aus. Die gleichzeitige oder überlappende Durchführung mehrerer Vorbereitungsmaßnahmen (wie z. B. Blut abnehmen) oder Aufklärungsgespräche ist jedoch zulässig und in der Regel sogar erwünscht (vgl. § 1.2.2).

Mit Hilfe der in § 2.3.3 eingeführten Schablone zur Beschreibung eines wechselseitigen Ausschlusses lassen sich diese Integritätsbedingungen für einen *einzelnen* Patienten durch den Graphen in Abb. 2.61 beschreiben, der besagt, daß der Patient zu jedem Zeitpunkt

- *entweder* für beliebig viele Untersuchungen vorbereitet werden kann
  (oberer Zweig des wechselseitigen Ausschlusses)
- *oder* für beliebig viele Untersuchungen aufgeklärt werden kann (mittlerer Zweig)
- *oder* sich in genau einer Untersuchungsstelle befinden kann (unterer Zweig).

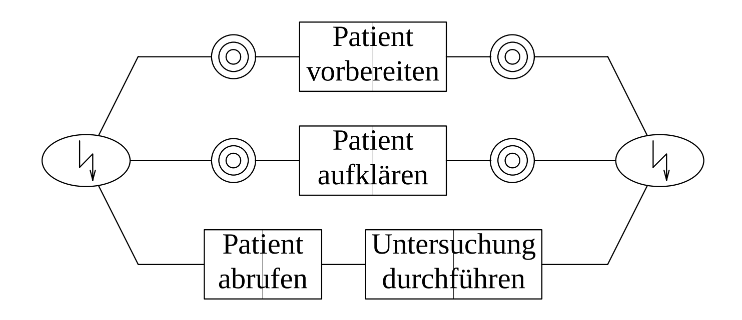


Abbildung 2.61: Integritätsbedingung für einen einzelnen Patienten

2.7.1.2 Bedingung für alle Patienten

Ebenso wie sich die Aktionen in einem Kopiergeschäft immer auf einen bestimmten Kopierer beziehen und daher in § 2.3.4 mit einem Parameter k versehen wurden, beziehen sich die Aktivitäten in Abb. 2.61 sowohl auf einen bestimmten Patienten p als auch auf eine bestimmte Untersuchungsart bzw. -stelle u. Sie sollten daher ebenfalls mit entsprechenden Parametern versehen werden, die es erlauben, über Aktivitäten wie z. B. Patient p für Untersuchung u abrufen oder Untersuchung u für Pa-

[10] Vergleiche die Beispiel-Workflows in Abb. 1.1 und 1.2, § 1.1.5.

tient p durchführen zu sprechen. Bei den Werten des Parameters p handelt es sich typischerweise um *Patienten-Identifikationsnummern*, während der Parameter u *symbolische Konstanten*, wie z. B. sono für Sonographie oder endo für Endoskopie, enthalten kann.

Durch geeignete *Quantifizierung* dieser Parameter mit Hilfe von Für-ein- und Für-alle-Verzweigungen erhält man den Graphen in Abb. 2.62: Die Für-alle-Verzweigung über p beschreibt, daß die durch ihren Rumpf spezifizierte Bedingung für *jeden* Patienten p separat einzuhalten ist, Aktivitäten *verschiedener* Patienten jedoch *unabhängig* voneinander ausgeführt werden können. Der Rumpf dieser Verzweigung entspricht im wesentlichen dem Graphen aus Abb. 2.61, der in geeigneter Weise um Für-ein-Verzweigungen erweitert wurde.

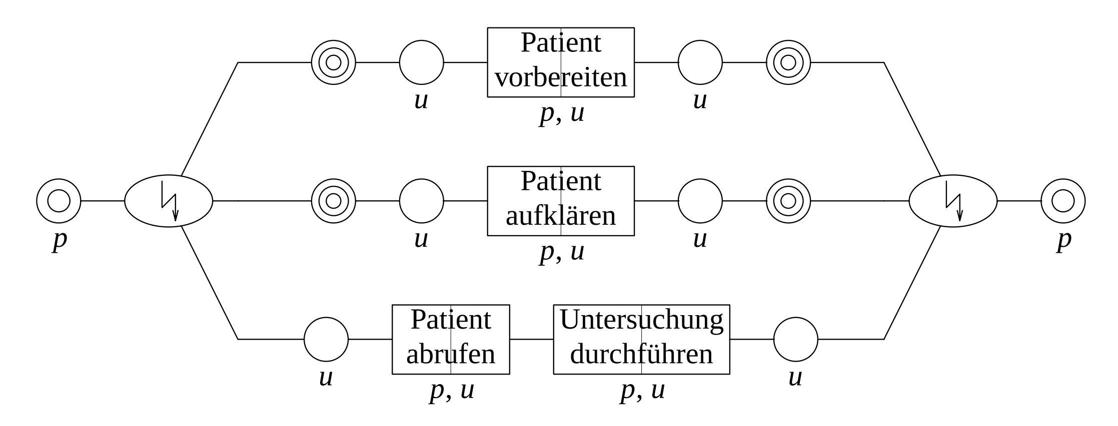


Abbildung 2.62: Integritätsbedingung für alle Patienten

2.7.1.3 Korrekte Verwendung von Für-ein-Verzweigungen

Wie bereits in § 2.3.4.6 angedeutet wurde, muß beim Einsatz dieser Verzweigungen sorgfältig auf ein korrektes „Scoping" geachtet werden, damit die Semantik des formulierten Graphen wirklich der intendierten Bedeutung entspricht. Beispielsweise müssen die Für-ein-Verzweigungen in den oberen beiden Zweigen des wechselseitigen Ausschlusses jeweils *innerhalb* der Beliebig-oft-Verzweigungen plaziert werden, damit *jede* Gruppe, die den Rumpf einer solchen Verzweigung durchläuft, jeweils eine *beliebige* Belegung des Parameters u wählen kann, die typischerweise *verschieden* von den Belegungen der übrigen Gruppen ist. Hätte man die Verschachtelung der beiden Verzweigungsarten jeweils vertauscht (vgl. Abb. 2.63), so müßte jeweils eine *gesamte* Beliebig-oft-Verzweigung mit einer

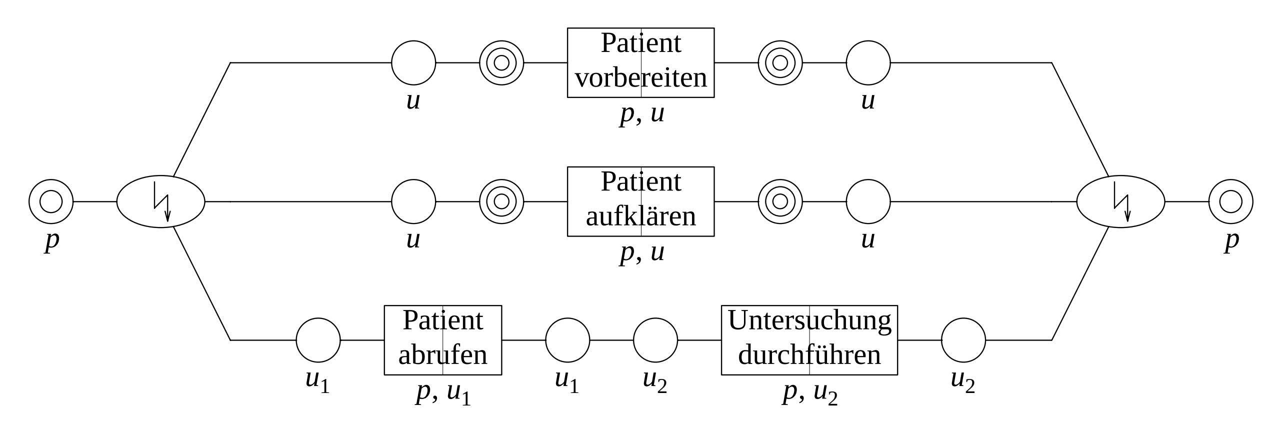


Abbildung 2.63: Fehlerhafte Verwendung von Für-ein-Verzweigungen

einzigen Belegung des Parameters *u* durchlaufen werden, d. h. man könnte den Patienten *p* zwar beliebig oft gleichzeitig für *dieselbe* Untersuchung *u* vorbereiten oder aufklären (was natürlich nicht sehr sinnvoll ist), aber nicht − wie eigentlich beabsichtigt − mehrere Vorbereitungsmaßnahmen bzw. Aufklärungsgespräche für *verschiedene* Untersuchungen gleichzeitig durchführen.

Ebenso ist darauf zu achten, daß sich die Sequenz Patient abrufen − Untersuchung durchführen im Wirkungsbereich (engl. scope) einer *einzigen* Für-ein-Verzweigung befindet, damit sichergestellt ist, daß beide Aktivitäten mit *derselben* Belegung des Parameters *u* durchlaufen werden. Hätte man jede dieser Aktivitäten mit einer *eigenen* Für-ein-Verzweigung umgeben (vgl. Abb. 2.63), so wäre es aus Sicht des Graphen beispielsweise zulässig, einen Patienten *p* zur Sonographie abzurufen (Belegung u_1 = sono), anschließend aber eine Endoskopie für ihn durchzuführen (Belegung u_2 = endo).

2.7.2 Integritätsbedingungen für Untersuchungsstellen

Nachdem mit Hilfe der Integritätsbedingung für Patienten (Abb. 2.62) sichergestellt ist, daß sich kein Patient während seines Klinikaufenthalts „zerreißen" muß (um an mehreren Stellen gleichzeitig zu sein), sollen im folgenden einige Bedingungen entwickelt werden, die eine Überlastung von Untersuchungsstellen ausschließen.

2.7.2.1 Allgemeine Kapazitätsbeschränkung

Der Graph in Abb. 2.64 beschreibt zum Beispiel, daß in jeder Untersuchungsstelle *u* maximal fünf Patienten *p* gleichzeitig untersucht werden können.

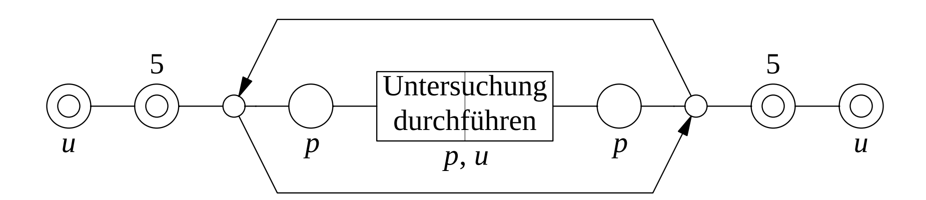


Abbildung 2.64: Allgemeine Kapazitätsbeschränkung

Um die Wirkungsweise eines solchen Graphen zu verstehen, ist es meist sinnvoll, ihn *von innen nach außen* zu entwickeln:

- Die Für-ein-Verzweigung $\underset{p}{\bigcirc} \cdots \underset{p}{\bigcirc}$ erlaubt die Durchführung *einer* Untersuchung *u* für einen beliebigen Patienten *p*.
- Durch die umgebende Wiederholung wird ausgedrückt, daß sich dieser Vorgang beliebig oft mit beliebig wechselnden Patienten *p* wiederholen kann.
- Durch die Mehrfach-Verzweigung $\overset{5}{\circledcirc} \cdots \overset{5}{\circledcirc}$ wird als nächstes festgelegt, daß fünf derartige Wiederholungen parallel ausgeführt werden dürfen, d. h. daß bis zu fünf Patienten gleichzeitig Untersuchung *u* durchlaufen können.
- Durch die Für-alle-Verzweigung $\underset{u}{\circledcirc} \cdots \underset{u}{\circledcirc}$ schließlich wird diese Bedingung für alle Untersuchungsarten bzw. -stellen *u* separat eingehalten.

2.7.2.2 Spezielle Kapazitätsbeschränkungen

Anstelle oder auch zusätzlich zu dieser *allgemeinen* Kapazitätsbeschränkung ist es möglich, *spezielle* (typischerweise restriktivere) Bedingungen für einzelne Untersuchungsstellen zu formulieren, indem

man die Für-alle-Verzweigung über *u* wegläßt und den Parameter *u* stattdessen durch einen konkreten Wert ersetzt. Abbildung 2.65 zeigt dies exemplarisch für die Sonographie (sono), in der maximal drei Untersuchungen gleichzeitig durchgeführt werden sollen.

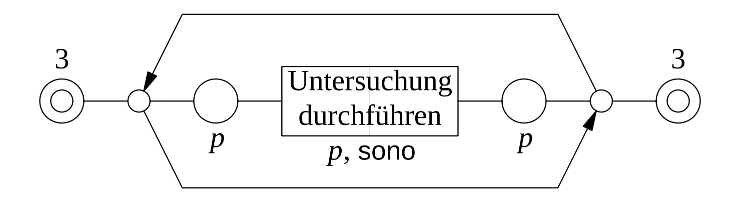


Abbildung 2.65: Spezielle Kapazitätsbeschränkung für die Sonographie

2.7.2.3 Kopplung parametrisierter Graphen

Koppelt man die allgemeine Bedingung aus Abb. 2.64 mit dieser speziellen Bedingung (vgl. Abb. 2.66), so dürfen Aktionen, die beiden Zweigen der Kopplung gemeinsam sind, nur ausgeführt werden, wenn sie in beiden Zweigen gleichzeitig durchlaufen werden können (vgl. § 2.3.2.2). Hierbei stellt sich allerdings die Frage, welche Aktionen zu einem (Teil-)Graphen gehören, dessen Aktivitäten parametrisiert sind und der Für-ein- oder Für-alle-Verzweigungen enthält.

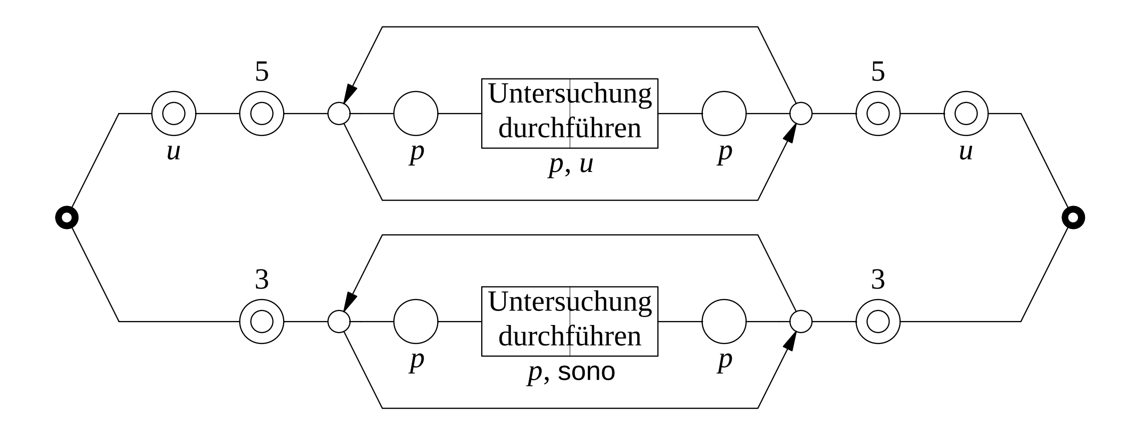


Abbildung 2.66: Kopplung von allgemeiner und spezieller Kapazitätsbeschränkung

Interpretiert man eine Für-ein- bzw. Für-alle-Verzweigung über einen Parameter *p* als unendliche Entweder-oder- bzw. Sowohl-als-auch-Verzweigung, bei der der Parameter *p* alle prinzipiell denkbaren Werte durchläuft, so läßt sich diese Frage relativ leicht beantworten: Der obere Zweig der Kopplung enthält die Menge aller Aktivitäten Untersuchung *u* für Patient *p* durchführen für *alle* denkbaren Belegungen der Parameter *p* und *u*, während der untere Zweig lediglich die Aktivitäten Untersuchung sono für Patient *p* durchführen für alle denkbaren Belegungen des Parameters *p* enthält.

Trägt man die möglichen Werte der Parameter *p* und *u* (für die Aktivität Untersuchung durchführen) jeweils auf einer Achse eines Koordinatensystems auf, so entsteht ein zweidimensionaler *Parameterraum* wie in Abb. 2.67. In diesem Raum entspricht die erste Menge von Aktivitäten (oberer Zweig der Kopplung) der zweidimensionalen Menge aller (schwarzen und weißen) Punkte, während die zweite Menge (unterer Zweig) der eindimensionalen Menge der schwarzen Punkte entspricht. Diese zweite Menge stellt somit auch die Schnittmenge der beiden Mengen, d. h. die Menge der beiden

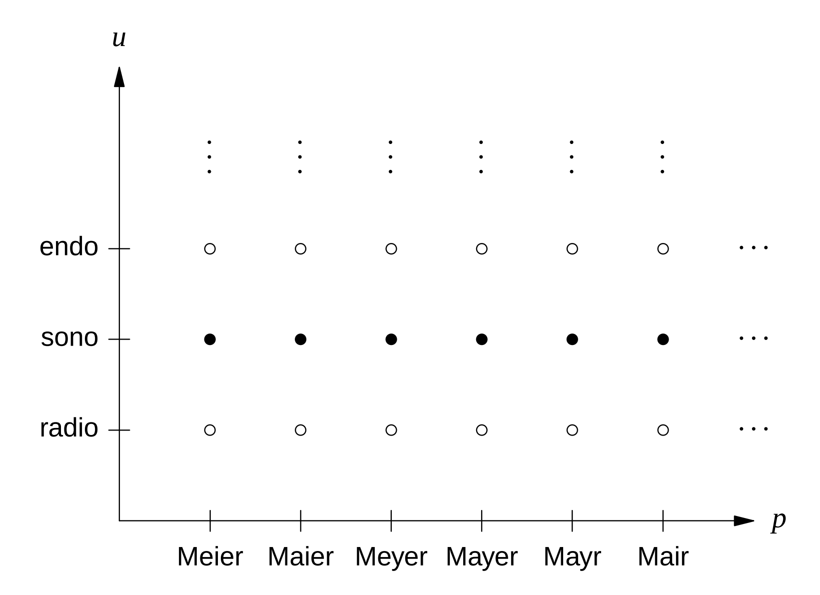


Abbildung 2.67: Zweidimensionaler Parameterraum

Zweigen gemeinsamen Aktivitäten dar, während die Menge der weißen Punkte der Menge von Aktivitäten entspricht, die nur im oberen Zweig der Kopplung auftreten.

Folglich darf eine Aktivität Untersuchung sono für Patient *p* durchführen nur dann ausgeführt werden, wenn sie sowohl im oberen als auch im unteren Zweig der Kopplung durchlaufen werden kann, was genau dann der Fall ist, wenn sich momentan höchstens zwei derartige Aktivitäten in Ausführung befinden. Sind bereits drei Sonographien aktiv, so würde der obere Zweig der Kopplung zwar die Ausführung weiterer zwei erlauben, der untere Zweig würde dies jedoch unterbinden.

Aktivitäten Untersuchung *u* für Patient *p* durchführen mit einer Parameterbelegung $u \neq$ sono hingegen kommen nur im oberen Zweig der Kopplung vor und dürfen daher ausgeführt werden, wenn sie von diesem zugelassen werden. Dies ist genau dann der Fall, wenn sich momentan höchstens vier derartige Aktivitäten in Ausführung befinden.

Anmerkung: In Abb. 2.67 wurden die Achsen des Koordinatensystems bewußt nur mit exemplarischen Werten beschriftet, die sinnvoll für die Parameter *p* bzw. *u* sind, obwohl beide Achsen grundsätzlich dieselbe Menge *aller* denkbaren Parameterwerte repräsentieren. Dementsprechend würde der Graph in Abb. 2.66 z. B. auch die unsinnige Aktivität Untersuchung Meier für Patient sono durchführen zulassen. Wie in § 2.3.4.3 erwähnt, sorgt jedoch das für die Ausführung der Untersuchungs-Workflows verantwortliche WfMS dafür, daß derartige Aktivitäten niemals ausgeführt werden können.

2.7.3 Begrenzung von Warteschlangen

In Krankenhäusern ist es übliche Praxis, Patienten *rechtzeitig* zu einer Untersuchung abzurufen und dann ggf. in der Untersuchungsstelle warten zu lassen, um so einen eventuellen Leerlauf zu vermeiden. Allerdings sollte die Anzahl der wartenden Patienten auf ein vernünftiges Maß begrenzt werden. Auch dies läßt sich mit einem geeigneten Interaktionsgraphen spezifizieren (vgl. Abb. 2.68): Ähnlich wie beim Graphen in Abb. 2.64 wird festgelegt, daß sich pro Untersuchungsstelle *u* maximal acht Patienten *p* innerhalb der Sequenz Patient *p* für Untersuchung *u* abrufen − Untersuchung *u* für Patient *p* durchführen befinden dürfen. Da fünf von ihnen i. d. R. gleichzeitig untersucht werden, befinden sich somit maximal drei in Wartestellung. (Für Untersuchungsstellen mit geringerer Kapazität als fünf sollten − analog zu den speziellen Kapazitätsbeschränkungen − auch spezielle „Warteschlangen-Begrenzungs-Bedingungen“ formuliert werden, in denen der Faktor 8 durch einen entsprechend niedrigeren Wert ersetzt wird.)

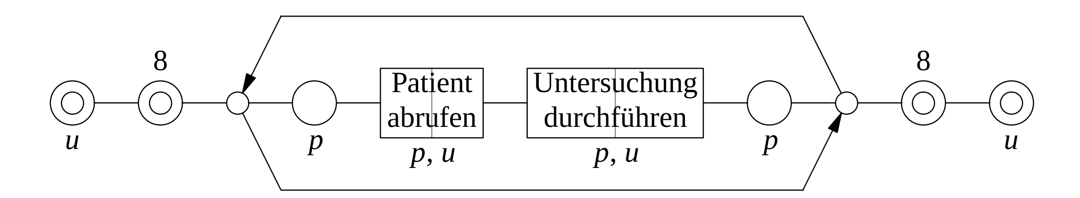


Abbildung 2.68: Begrenzung von Warteschlangen

2.7.4 Reihenfolgen und temporale Aspekte

2.7.4.1 Erste Formulierung

Gelegentlich ist für zwei verschiedene Untersuchungen desselben Patienten eine bestimmte *Reihenfolge* einzuhalten, weil andernfalls die eine Untersuchung das Ergebnis der anderen verfälschen würde. Beispielsweise sollte eine Sonographie immer *vor* einer Endoskopie durchgeführt werden, weil die bei der Endoskopie ggf. in den Darm gepumpte Luft eine anschließende sonographische Beurteilung des Bauchraums unmöglich macht (vgl. auch § 1.3.1.1).

Allerdings darf die umgangssprachliche Formulierung „Sonographie vor Endoskopie" aus verschiedenen Gründen nicht ganz wörtlich (d. h. als strikte Sequenz Sonographie − Endoskopie) interpretiert werden: Zum einen muß es möglich sein, jede der beiden Untersuchungen auch einzeln durchzuführen, d. h. nach der Durchführung einer Sonographie muß nicht notwendigerweise eine Endoskopie folgen (es könnte beispielsweise auch eine weitere Sonographie durchgeführt werden), noch muß einer Endoskopie immer eine Sonographie vorausgehen. Zum anderen darf eine Sonographie durchaus auch *nach* einer Endoskopie durchgeführt werden, sofern seit der Beendigung der (letzten) Endoskopie *ausreichend viel Zeit* verstrichen ist, in der sich der Zustand des Darms wieder normalisiert hat.

Dies bedeutet, daß die Untersuchungen Sonographie und Endoskopie für einen Patienten *p* beliebig oft in beliebiger Reihenfolge durchgeführt werden können, sofern nach einer Endoskopie jeweils eine bestimmte *Wartezeit* (z. B. 24 Stunden) eingehalten wird. Verwendet man zur graphischen Darstellung dieser Wartezeit eine *Stoppuhr*, so kann die genannte Bedingung durch den Graphen in Abb. 2.69 beschrieben werden.

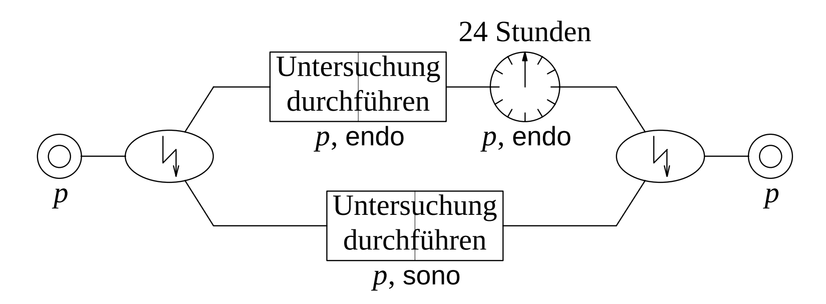


Abbildung 2.69: Mindestabstand zwischen Endoskopie und Sonographie (erste Formulierung)

2.7.4.2 Pseudo-Aktivitäten

Konzeptuell entspricht eine Stoppuhr einer *Pseudo-Aktivität*, die nicht von einem gewöhnlichen Benutzer, sondern von einem speziellen *Agenten* automatisch ausgeführt wird, sobald sie aus Sicht des Graphen zulässig ist. Die Stoppuhr in Abb. 2.69 (oder genauer: eine „Ausprägung" dieser Stoppuhr) wird also immer dann gestartet, wenn eine Aktivität Untersuchung endo für Patient *p* durchführen beendet wurde. Die Ausführungsdauer dieser Pseudo-Aktivität entspricht genau der als Parameter über-

gebenen Zeitspanne, im Beispiel also 24 Stunden. Das bedeutet, daß der obere Zweig des wechselseitigen Ausschlusses erst 24 Stunden nach Beendigung der Endoskopie vollständig durchlaufen ist und somit frühestens zu diesem Zeitpunkt die nächste Sonographie oder Endoskopie für diesen Patienten begonnen werden kann.

Anmerkung: Wird der Graph in Abb. 2.69 für mehrere Patienten *p* durchlaufen, so „ticken“ u. U. mehrere Stoppuhren für verschiedene Patienten gleichzeitig. Außerdem können auch für ein und denselben Patienten mehrere Uhren gleichzeitig laufen, wenn man analoge Graphen für andere Untersuchungsarten formuliert. Damit in einem solchen Szenario jede laufende Uhr eindeutig einem Patienten *p* und einer Untersuchungsart *u* zugeordnet werden kann, erhält die Pseudo-Aktivität Stoppuhr diese Angaben als zusätzliche Parameter.

2.7.4.3 Verbesserte Formulierung

Sofern nach einer Endoskopie keine Sonographie, sondern eine weitere Endoskopie durchgeführt werden soll, ist die genannte Wartezeit allerdings nicht erforderlich und könnte zu unnötigen Verzögerungen im Gesamtablauf führen. Abbildung 2.70 spezifiziert daher, daß die Sequenz Untersuchung durchführen − Stoppuhr beliebig oft gleichzeitig oder überlappend durchlaufen werden darf, d. h. daß eine weitere Endoskopie gestartet werden darf, während die Stoppuhr der letzten noch „tickt“. Die gesamte Beliebig-oft-Verzweigung im oberen Zweig des wechselseitigen Ausschlusses ist jedoch erst dann vollständig durchlaufen, wenn alle Gruppen, die ihren Rumpf durchlaufen, den rechten ◎-Knoten erreicht haben, was genau 24 Stunden nach Beendigung der *letzten* Endoskopie dieses Patienten der Fall ist. Frühestens dann kann also die nächste Sonographie für diesen Patienten begonnen werden, d. h. erst dann sollte er wieder zur Sonographie *abgerufen* werden können.

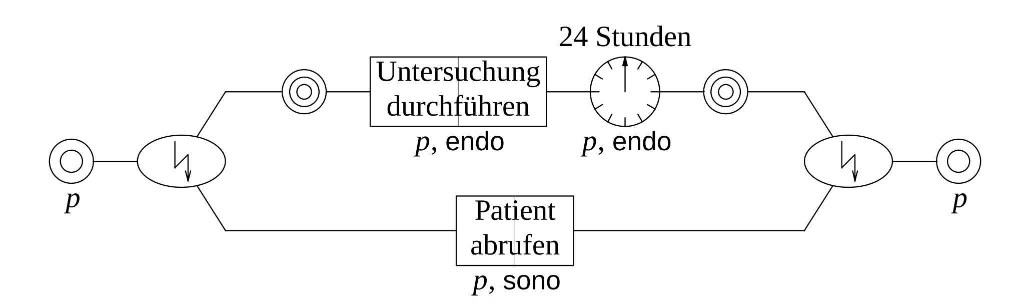


Abbildung 2.70: Mindestabstand zwischen Endoskopie und Sonographie (verbesserte Formulierung)

Isoliert betrachtet, würde dieser Graph sogar die *gleichzeitige* Durchführung mehrerer Endoskopien für einen Patienten erlauben. Diese unerwünschte Möglichkeit wird jedoch durch die Integritätsbedingung für Patienten (Abb. 2.62) unterbunden.

2.7.4.4 Definition als Abkürzung

Da der Graph in Abb. 2.70 eine auf den ersten Blick nicht unbedingt nachvollziehbare *Implementierung* der abstrakten Bedingung „Zwischen Endoskopie und Sonographie müssen mindestens 24 Stunden liegen“ darstellt, ist es ratsam, diese Implementierung in einer Abkürzung zu *verbergen*, in der man auch von den konkreten Werten endo, sono und 24 Stunden abstrahieren kann (vgl. Abb. 2.71). Auf diese Weise ist es auch für einen ungeübten Anwender möglich, Mindestabstände zwischen beliebigen Untersuchungsarten einfach und auf den ersten Blick verständlich zu formulieren (vgl. Abb. 2.72). In gleicher Weise könnte man auch die Graphen zur Formulierung von Kapazitätsbeschränkungen und zur Begrenzung von Warteschlangen in entsprechenden Abkürzungen verbergen.

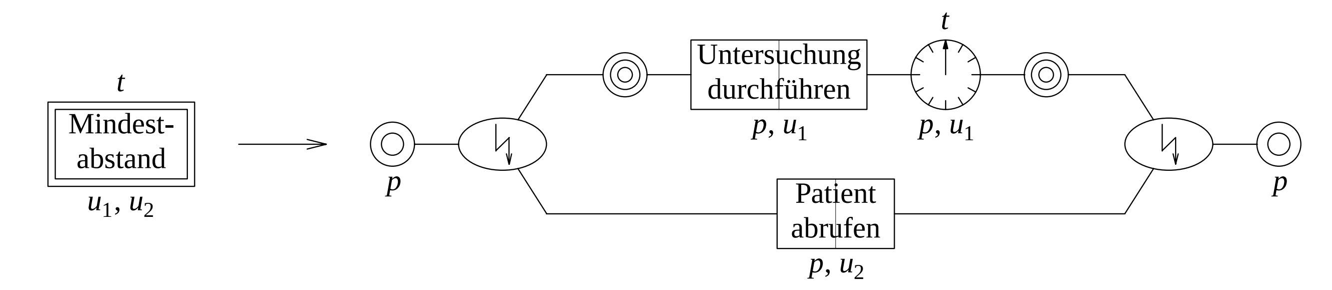


Abbildung 2.71: Mindestabstand als parametrisierte Abkürzung



Abbildung 2.72: Anwendung der Abkürzung Mindestabstand

2.7.5 Zusammenfassung der Bedingungen

Für einen reibungslosen Ablauf in einer Klinik ist es erforderlich, daß alle Bedingungen, die in diesem Abschnitt durch Interaktionsgraphen spezifiziert wurden (und vermutlich noch weitere), gleichzeitig eingehalten werden. Hierfür müssen die einzelnen Graphen prinzipiell mit Hilfe einer Kopplung verknüpft werden, damit eine Aktivität dann und nur dann ausgeführt werden darf, wenn sie in allen Graphen, in denen sie auftritt, gleichzeitig durchlaufen werden kann (vgl. § 2.3.2.2), d. h. wenn alle Bedingungen, die eine *Aussage* über diese Aktivität machen, ihre Ausführung erlauben. Aus Gründen der besseren Übersichtlichkeit und Handhabbarkeit wird eine Menge *unverknüpfter* Graphen (vgl. Abb. 2.73) jedoch implizit mittels einer Kopplung verknüpft, so daß man auf eine explizite Verknüpfung der einzelnen Graphen verzichten kann.

2.8 Zusammenfassung

2.8.1 Operatoren

Tabelle 2.74 zeigt alle Operatoren von Interaktionsgraphen − zusammen mit den Abschnitten, in denen sie eingeführt wurden − noch einmal im Zusammenhang und kann daher als Kurzreferenz verwendet werden. Außerdem werden hier bereits die im folgenden Kapitel eingeführten formalen Bezeichnungen der Operatoren erwähnt.

2.8.2 Prinzipien

Auch die in diesem Kapitel explizit oder implizit erwähnten Grundprinzipien von Interaktionsgraphen sollen im folgenden noch einmal kurz rekapituliert werden.

2.8.2.1 Orthogonalität

Mit dem Begriff *Orthogonalität* bezeichnet man gewöhnlich die Möglichkeit, die von einem Formalismus angebotenen Operatoren *uneingeschränkt* miteinander *kombinieren* zu können. Übertragen auf Interaktionsgraphen bedeutet das, daß die verschiedenen Verzweigungen, Wiederholungen etc. *belie-*

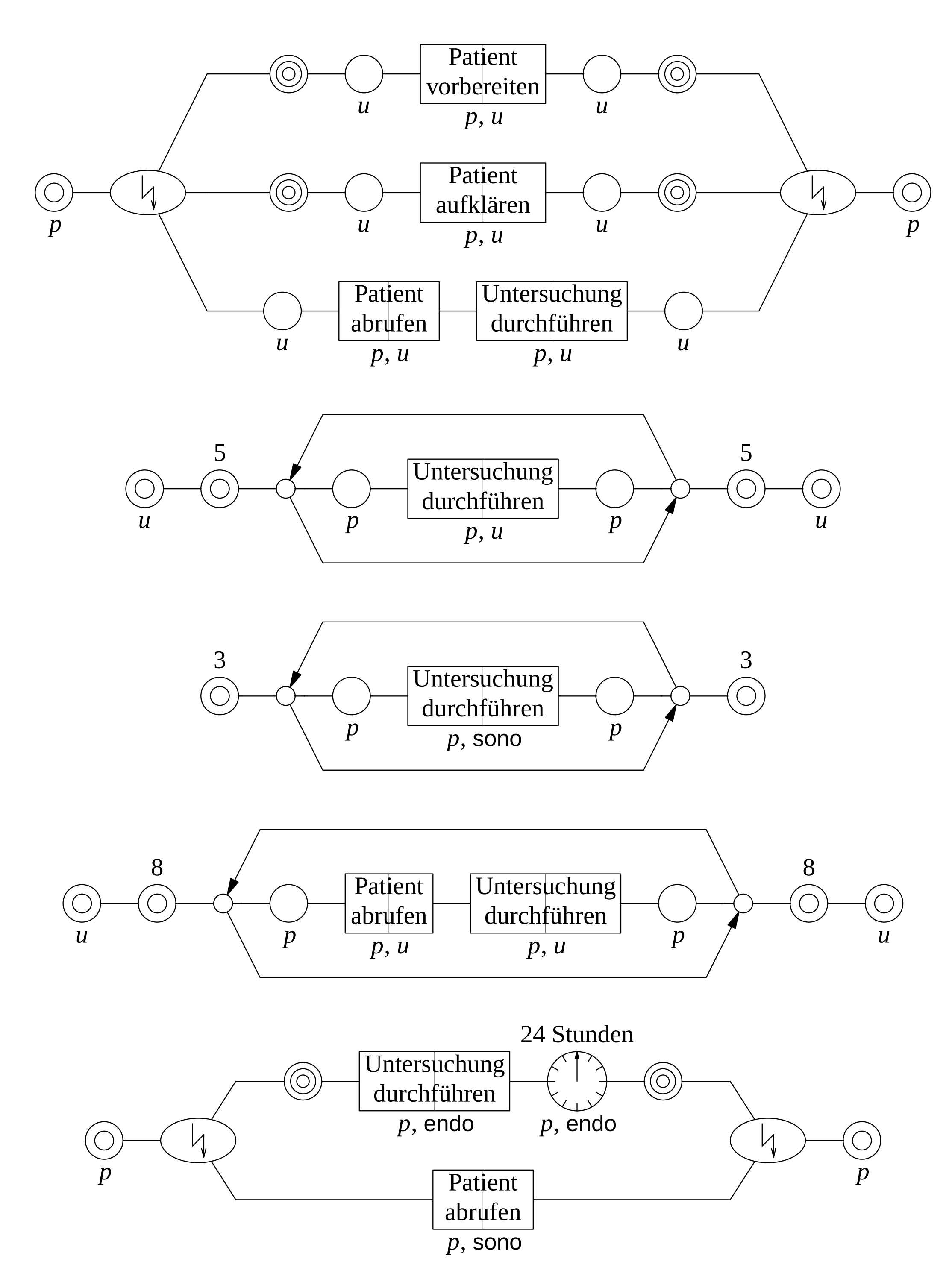


Abbildung 2.73: Zusammenfassung der Bedingungen

big verschachtelt werden dürfen, oder anders ausgedrückt, daß an jeder Stelle eines Graphen, an der eine atomare Aktion stehen darf, auch ein *beliebig komplexer Teilgraph* stehen darf.

Eigentlich sollte diese Eigenschaft so selbstverständlich sein, daß sie gar nicht explizit erwähnt werden muß. Insbesondere stellt sie eine *conditio sine qua non* für die konsequente Anwendbarkeit von *Abstraktionsmechanismen* (Abkürzungen und Schablonen; siehe § 2.8.2.2) und die *modulare Kombination* (§ 2.8.2.4) von Graphen dar. Und dennoch weisen viele verwandte Formalismen gerade in diesem Punkt erhebliche Schwächen auf, indem sie bestimmte Kombinationen von Operatoren ver-

Bezeichnung	*Graph*	*Abschnitt*
Sequenz *(sequentielle Komposition)*	y z	2.2.1 *(3.3.1.2)*
Entweder-oder-Verzweigung *(Disjunktion)*	y z	2.2.1 *(3.3.1.4)*
Sowohl-als-auch-Verzweigung *(parallele Komposition)*	y z	2.2.2.4 *(3.3.1.6)*
Kopplung *(Synchronisation)*	y z	2.3.2.2 *(3.3.1.10)*
Wiederholung *(sequentielle Iteration)*	y	2.2.5.1 *(3.3.1.3)*
Eventuell-Verzweigung *(Option)*	y	2.2.5.2 *(3.3.1.5)*
Mehrfach-Ausführung *(sequentieller Multiplikator)*	*n* y *n*	2.2.5.3 *(3.3.2.1)*
Beliebig-oft-Verzweigung *(parallele Iteration)*	y	2.3.1.3 *(3.3.1.7)*
Mehrfach-Verzweigung *(paralleler Multiplikator)*	*n* y *n*	2.2.3.3 *(3.3.2.1)*
Für-alle-Verzweigung *(paralleler Quantor)*	y *p* *p*	2.3.4.4 *(3.3.3.1)*
Für-ein-Verzweigung *(Disjunktions-Quantor)*	y *p* *p*	2.3.4.6 *(3.3.3.1)*
Abkürzung (Beispiel) *(Makro)*	Fünf Mark	2.2.4.4 *(3.3.2.2)*
Schablone (Beispiel) *(Makro)*	y z	2.3.3 *(3.3.2.2)*

Tabelle 2.74: Operatoren von Interaktionsgraphen

bieten oder manche Operatoren (z. B. Für-alle-Verzweigungen) nur auf der „äußersten Ebene“ zulassen (vgl. auch § 6.2.5.3).

2.8.2.2 Abstraktionsmechanismen

Abstraktion ist ganz allgemein ein unerläßliches konzeptuelles Hilfsmittel, um komplexe Problemstellungen bewältigen zu können. *Abkürzungen* und *Schablonen* stellen konkrete *Abstraktionsmechanismen* dar, die es erlauben, komplexe Interaktionsgraphen *schrittweise* (entweder von innen nach außen oder von außen nach innen) zu entwickeln, in überschaubare *Teilgraphen* zu zerlegen und diese in verschiedenen Kontexten *wiederzuverwenden*. Außerdem ist es möglich, zwischen abstrakten *Konzepten* (wie z. B. wechselseitiger Ausschluß) und konkreten *Implementierungen* der Konzepte (z. B. Wiederholung einer Entweder-oder-Verzweigung; vgl. § 2.3.3.1) zu trennen, wobei letztere bei Bedarf von einem erfahrenen „Experten“ erstellt werden können, während erstere (d. h. konkrete Abkürzungen oder Schablonen) auch von weniger geübten Anwendern − ohne Kenntnis ihrer Implementierung − verwendet werden können.

2.8.2.3 Erweiterbarkeit

Da Abkürzungen und Schablonen *integrale Bestandteile* von Interaktionsgraphen darstellen, die sich *nahtlos* in das Gesamtkonzept einfügen, ist der Formalismus *homogen erweiterbar*. Beispielsweise können Abkürzungen in derselben Weise wie Aktionen Parameter erhalten und Schablonen in derselben Weise wie „eingebaute“ Operatoren verwendet werden. Auf diese Weise ist es unter anderem möglich, *benutzerdefinierte Operatoren* (wie z. B. die Blitz-Verzweigung in § 2.3.3.1) zu definieren − eine Möglichkeit, die man nur in wenigen Formalismen oder Sprachen vorfindet.

2.8.2.4 Modulare Kombination

Aufgrund der vollständigen Orthogonalität des Formalismus können unabhängig voneinander entwickelte Teilgraphen beliebig zu neuen Graphen zusammengefügt werden. Insbesondere erlaubt der *Kopplungsoperator*, der von vielen verwandten Formalismen nicht oder nur ansatzweise unterstützt wird, eine schrittweise (oder auch nachträgliche) *Einschränkung* von Integritätsbedingungen, ohne hierfür die vorliegenden Graphen *modifizieren* zu müssen. Auch die Tragweite und Bedeutung dieser Aussage weiß man vermutlich nur dann richtig einzuschätzen, wenn man „schlechte Erfahrungen“ mit anderen Formalismen gemacht hat (vgl. z. B. § 6.2.3.4).

2.8.2.5 Explizite Iterationen

Würde man rekursive oder zyklische Abkürzungsdefinitionen erlauben, so könnte man prinzipiell auf Wiederholungen und Beliebig-oft-Verzweigungen verzichten, da diese mittels *Rekursionsgleichungen* auf Sequenzen bzw. Sowohl-als-auch-Verzweigungen zurückgeführt werden können (vgl. auch § 3.4.10). Zur Illustration dieser Aussage betrachte man die Graphen in Abb. 2.75 und 2.76, die äquivalent zu Abb. 2.21 (§ 2.2.5.1) bzw. Abb. 2.30 (§ 2.3.1.4) sind. Allerdings zeigen bereits diese beiden einfachen Beispiele, daß Graphen mit *expliziten* Iterationen[11] sowohl *kompakter* als auch *leichter verständlich* sind, weil sie *direkt* − ohne Umweg über eine Rekursionsgleichung − die intendierte Semantik zum Ausdruck bringen.

Ausgehend von diesem Standpunkt, kann man für die allermeisten Anwendungen vollständig auf Rekursion verzichten, da „inhärent rekursive Probleme“ (d. h. Bedingungen, die sich nur mit Hilfe von Rekursionsgleichungen beschreiben lassen) äußerst selten sind (vgl. auch § 3.5.4). Aus diesem Grund − und da die Hinzunahme von Rekursion die ohnehin schon umfangreiche und komplexe formale Behandlung und Implementierung von Interaktionsgraphen weiter verkomplizieren würde − sind rekursive Definitionen von Interaktionsgraphen bewußt ausgeschlossen.

[11] Wiederholung und Beliebig-oft-Verzweigung werden in Kapitel 3 auch als sequentielle bzw. parallele *Iteration* bezeichnet.

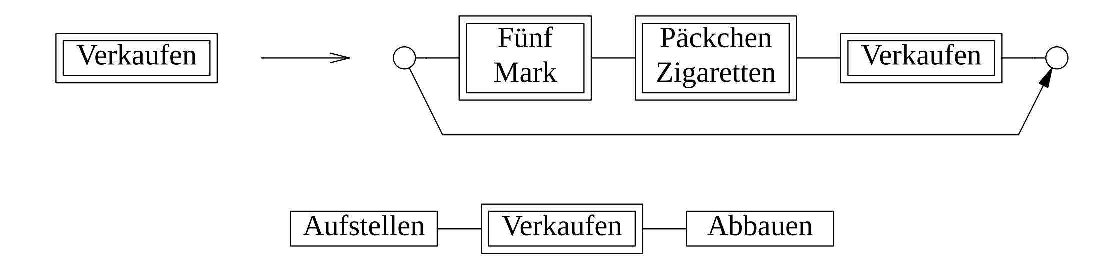


Abbildung 2.75: Wiederholung mittels Rekursion

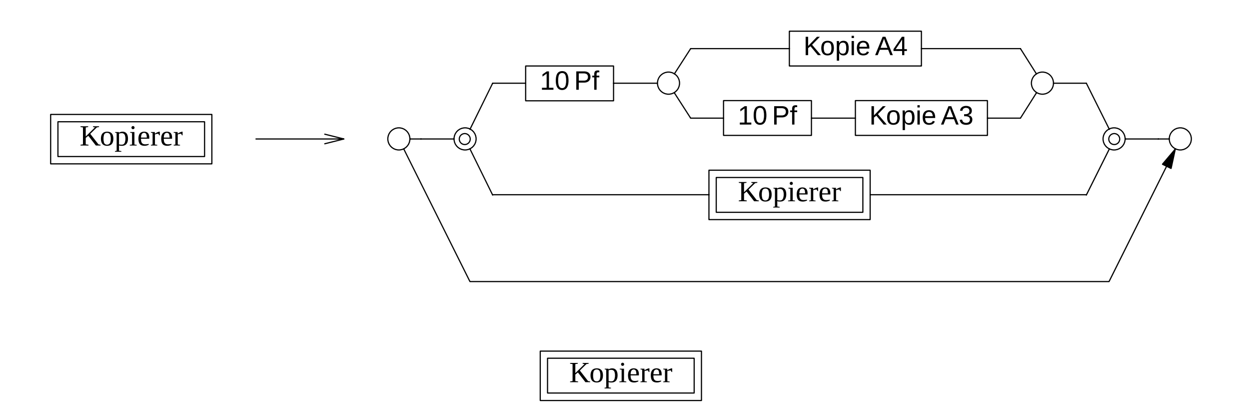


Abbildung 2.76: Beliebig-oft-Verzweigung mittels Rekursion

2.8.2.6 Deterministisches Verhalten

Wie in § 2.4.3 bereits ausführlich erläutert wurde, dürfen bei der Interpretation eines Graphen *keine nichtdeterministischen Entscheidungen* gefällt werden, die dazu führen, daß prinzipiell zulässige Wege vorzeitig „abgeschnitten“ werden. Stattdessen müssen in einer Konfliktsituation *alle* möglichen Alternativen so lange weiterverfolgt werden, bis sie sich definitiv als falsch erweisen.

Kapitel 3

Interaktionsausdrücke

3.1 Einleitung

3.1.1 Motivation

Im vorangegangenen Kapitel wurde eine *zulässige Ausführungsreihenfolge* eines Interaktionsgraphen anschaulich als Folge *passierter Aktionen* definiert, die man erhält, wenn man den Graphen nach bestimmten Regeln von links nach rechts *durchläuft* (vgl. § 2.1.1). Die Menge *aller* zulässigen Ausführungsreihenfolgen erhält man dementsprechend, wenn man den Graphen auf *allen möglichen Wegen* durchläuft und jeweils die Folge der passierten Aktionen protokolliert (vgl. § 2.4.1).

Obwohl diese anschauliche Interpretation, zusammen mit einer gewissen praktischen Erfahrung im Umgang mit dem Formalismus, in der Regel genügt, um für konkrete Problemstellungen adäquate Lösungen zu formulieren, bleibt aus theoretischer Sicht doch eine gewisse Unsicherheit bezüglich der *exakten Bedeutung* mancher Graphen, insbesondere im Kontext *unendlicher Graphen* (wie z. B. Füralle-Verzweigungen) oder Graphen mit Sackgassen.

Darüber hinaus sind die anschaulich formulierten Traversierungsregeln für Graphen nur bedingt zur Verifikation *formaler Eigenschaften*, wie z. B. Kommutativität, Assoziativität oder Idempotenz bestimmter Operatoren, geeignet, und schließlich benötigt man auch zur Entwicklung einer *korrekten Implementierung* eines Formalismus einen „genormten Maßstab“, der (wie der Duden für die deutsche Rechtschreibung) „maßgebend in allen Zweifelsfällen“ ist.

Aus diesen Gründen werden im vorliegenden Kapitel *Interaktionsausdrücke* als *mathematischer Formalismus* mit einer exakt definierten *Semantik* eingeführt, der konzeptionell *äquivalent* zu Interaktionsgraphen ist, d. h. der zu jedem in Kapitel 2 eingeführten graphischen Operator einen entsprechenden mathematischen Operator anbietet und umgekehrt. Formal betrachtet, sind Interaktionsgraphen dann nur noch eine *alternative Darstellungsform* für Interaktionsausdrücke, ebenso wie Syntaxdiagramme lediglich eine graphische Repräsentation kontextfreier Grammatiken darstellen.

3.1.2 Überblick

Nach einer kurzen Einführung bzw. Rekapitulation der benötigten Grundlagen (§ 3.2), werden Interaktionsausdrücke in § 3.3 definiert und ihre Semantik mit Hilfe *formaler Sprachen* spezifiziert. Bei der Vorstellung der einzelnen Operatoren werden jeweils Parallelen zu den entsprechenden graphischen Operatoren aus Kapitel 2 gezogen und konkrete Beispiele der Abschnitte 2.2, 2.3 und 2.7 zur Illustration verwendet.

Im Anschluß daran werden in § 3.4 zahlreiche formale *Eigenschaften*, wie z. B. Kommutativität, Assoziativität und Idempotenz bestimmter Operatoren, formuliert und verifiziert. Zum Abschluß des Kapitels werden Interaktionsausdrücke in § 3.5 bezüglich ihrer *Ausdrucksmächtigkeit* untersucht und mit den beiden „prominentesten Vertretern“ der Chomsky-Hierarchie, regulären Ausdrücken und kontextfreien Grammatiken, verglichen.

3.2 Grundbegriffe und Bezeichnungen

3.2.1 Abstrakte und konkrete Aktionen

3.2.1.1 Grundmengen

Gegeben seien die folgenden *Grundmengen*:

- eine Menge Λ von *Aktionsnamen* (wie z. B. Kopie A4 oder Patient vorbereiten);
- eine Menge Π von *formalen Parametern* (wie z. B. k, p oder u);[1]
- eine *unendliche* Menge Ω von *Parameterwerten* (wie z. B. X5, Meier oder sono).

Typischerweise sind diese drei Mengen *paarweise disjunkt*, obwohl formal die Einschränkung $\Pi \cap \Omega = \emptyset$ ausreichend ist. Da die tatsächliche Kardinalität der Wertemenge Ω in praktischen Anwendungen meist nicht bekannt ist, wird sie vorsichtshalber als „unerschöpflich" angenommen.

3.2.1.2 Abstrakte Aktionen

Eine *abstrakte Aktion* $a = [a_0, a_1, \ldots, a_n]$ ist ein $(n{+}1)$-Tupel, bestehend aus einem Aktionsnamen $a_0 \in \Lambda$ und einer Folge $a_1, \ldots, a_n$ von Parametern oder *Argumenten*, bei denen es sich entweder um formale Parameter $\in \Pi$ oder um konkrete Werte $\in \Omega$ handelt. Für $n = 0$ erhält man eine *parameterlose Aktion* $a = [a_0]$, während man für $n \in \mathbb{N}$ eine *parametrisierte Aktion* erhält.

Die Menge

$$\Gamma = \{\, [a_0, a_1, \ldots, a_n] \mid n \in \mathbb{N}_0,\ a_0 \in \Lambda,\ a_1, \ldots, a_n \in \Pi \cup \Omega \,\}$$

bezeichnet die Menge aller abstrakten Aktionen.

3.2.1.3 Konkrete Aktionen

Eine *konkrete Aktion* $a = [a_0, a_1, \ldots, a_n]$ ist eine abstrakte Aktion, bei der alle Argumente $a_1, \ldots, a_n$ konkrete Werte $\in \Omega$ darstellen. Dementsprechend bezeichnet die Menge

$$\Sigma = \{\, [a_0, a_1, \ldots, a_n] \mid n \in \mathbb{N}_0,\ a_0 \in \Lambda,\ a_1, \ldots, a_n \in \Omega \,\}$$

die Menge aller konkreten Aktionen.

3.2.1.4 Anmerkungen

Da die in einem Interaktionsgraphen oder -ausdruck verwendeten Aktionen (wie z. B. Patient p für Untersuchung sono vorbereiten) formale Parameter enthalten können, handelt es sich hierbei um abstrakte Aktionen aus der Menge Γ. Bei konkret ausgeführten Aktionen hingegen müssen alle Argumente mit konkreten Werten belegt sein (z. B. Patient Meier für Untersuchung sono vorbereiten), d. h. diese Aktionen müssen aus der Menge Σ der konkreten Aktionen stammen.

Zur Vereinfachung der Notation können parameterlose Aktionen $[a_0]$ auch als a_0 und parametrisierte Aktionen $[a_0, a_1, \ldots, a_n]$ in der Form $a_0(a_1, \ldots, a_n)$ geschrieben werden.

[1] Nach der Einführung von Quantorausdrücken in § 3.3.3 werden formale Parameter auch als *Quantorparameter* bezeichnet.

3.2.2 Worte, Wortmengen und zugehörige Operationen

3.2.2.1 Worte

Ein *Wort* $w = \langle w_1, \ldots, w_n \rangle$ der Länge $|w| = n$ ist eine Folge von n konkreten Aktionen $w_1, \ldots, w_n \in \Sigma$. Das Wort $\langle\rangle$ der Länge 0 wird auch als *leeres Wort* bezeichnet.[2] Wie allgemein üblich, wird mit

$$\Sigma^* = \{ \langle w_1, \ldots, w_n \rangle \mid n \in \mathbb{N}_0, w_1, \ldots, w_n \in \Sigma \}$$

die Menge aller Worte über dem *Alphabet* Σ bezeichnet. Entsprechend bezeichnet

$$A^* = \{ \langle w_1, \ldots, w_n \rangle \mid n \in \mathbb{N}_0, w_1, \ldots, w_n \in A \}$$

die Menge aller Worte über einer beliebigen Teilmenge $A \subseteq \Sigma$.

Anmerkung: Da es sich bei den Aktionen w_i eines Wortes w per Definition um *konkrete* Aktionen handelt, entspricht ein Wort einer konkret ausgeführten Aktionsfolge.

3.2.2.2 Konkatenation und sequentielle Hülle

Die *Konkatenation* zweier *Worte* $u = \langle u_1, \ldots, u_m \rangle$ und $v = \langle v_1, \ldots, v_n \rangle$ (gelegentlich auch als *Produkt* von u und v bezeichnet) ist definiert als das Wort

$$u\,v = \langle u_1, \ldots, u_m, v_1, \ldots, v_n \rangle$$

der Länge $|u\,v| = m + n = |u| + |v|$.

Die *Konkatenation* zweier *Wortmengen* $U, V \subseteq \Sigma^*$ ist definiert als die Menge aller Worte $u\,v$, die man durch Konkatenation eines Wortes $u \in U$ mit einem Wort $v \in V$ erhält:

$$U\,V = \{ u\,v \mid u \in U, v \in V \}.$$

Durch mehrfache Konkatenation einer Menge $U \subseteq \Sigma^*$ mit sich selbst, erhält man die n-te *Potenz* der Menge U, die induktiv wie folgt definiert ist:

$$U^n = \begin{cases} \{ \langle\rangle \} & \text{für } n = 0, \\ U^{n-1}\,U & \text{für } n > 0. \end{cases}$$

Vereinigt man sämtliche Potenzen der Menge U, so erhält man die *sequentielle* (oder Kleenesche) *Hülle* von U:

$$U^* = \bigcup_{n=0}^{\infty} U^n = \{ u_1 \ldots u_n \mid n \in \mathbb{N}_0, u_1, \ldots, u_n \in U \}.$$

3.2.2.3 Verschränkung und parallele Hülle

Die *Verschränkung* (engl. shuffle) zweier Worte $u, v \in \Sigma^*$ ist definiert als:

$$u \otimes v = \{ u_1\,v_1 \ldots u_n\,v_n \mid n \in \mathbb{N}, u_1, v_1, \ldots, u_n, v_n \in \Sigma^*, u_1 \ldots u_n = u, v_1 \ldots v_n = v \}$$

[Ogden78, Shaw78, Guo96].

[2] In der Literatur über formale Sprachen werden Worte häufig in der Form $w = w_1 \ldots w_n$ (ohne umgebende Klammern und trennende Kommata) notiert, was zur Folge hat, daß rein „syntaktisch“ nicht zwischen einer Aktion $w \in \Sigma$ und einem Wort $w \in \Sigma^*$ der Länge 1 unterschieden werden kann. Um diesen Unterschied jedoch hervorzuheben und bei längeren Aktionsbezeichnern die Lesbarkeit zu verbessern, wird in dieser Arbeit die Notation $w = \langle w_1, \ldots, w_n \rangle$ bevorzugt, bei der die einzelnen Aktionen eines Worts durch Kommata getrennt werden und das ganze Wort in spitze Klammern eingeschlossen wird.
Konsequenterweise kann dann das leere Wort einfach als leeres Klammerpaar $\langle\rangle$ dargestellt werden, während andernfalls ein Ersatzsymbol wie z. B. λ oder ε benötigt wird. (Dieses wird meist auch zur Darstellung eines leeren *Ausdrucks* verwendet, was zu einer weiteren syntaktischen Überladung von Symbolen führt.)

Man beachte hierbei, daß $u_1, v_1, \ldots, u_n, v_n$ nicht einzelne Aktionen $\in \Sigma$, sondern beliebig lange *Teilworte* $\in \Sigma^*$ darstellen. Insbesondere können u_1 und v_n auch leer sein, d. h. ein Wort $u_1\, v_1 \ldots u_n\, v_n \in u \otimes v$ kann sowohl mit einem Teilwort von u als auch mit einem Teilwort von v beginnen bzw. enden. Man beachte außerdem, daß die Verschränkung zweier Worte (im Gegensatz zu ihrer Konkatenation) nicht ein einzelnes Wort, sondern eine *Menge* von Worten darstellt. Analog zur Konkatenation gilt jedoch, daß jedes Wort $w \in u \otimes v$ die Länge $|w| = |u| + |v|$ besitzt.

Analog zur Konkatenation zweier Wortmengen $U, V \subseteq \Sigma^*$ ist ihre Verschränkung definiert als die Menge aller Worte, die man durch Verschränkung eines Wortes $u \in U$ mit einem Wort $v \in V$ erhält:

$$U \otimes V = \bigcup_{\substack{u \in U \\ v \in V}} u \otimes v = \{ \, w \in \Sigma^* \mid \exists\, u \in U, v \in V\colon w \in u \otimes v \, \}.$$

Analog zur n-ten Potenz einer Menge $U \subseteq \Sigma^*$ ist die n-fache Verschränkung von U mit sich selbst definiert:

$$\bigotimes^{n} U = \begin{cases} \{ \langle\rangle \} & \text{für } n = 0, \\ \left(\bigotimes^{n-1} U \right) \otimes U & \text{für } n > 0. \end{cases}$$

Analog zur sequentiellen Hülle einer Menge $U \subseteq \Sigma^*$ ist schließlich auch ihre *parallele Hülle* definiert:

$$U\# = \bigcup_{n=0}^{\infty} \bigotimes^{n} U = \bigcup_{\substack{n \in \mathbb{N}_0 \\ u_1, \ldots, u_n \in U}} u_1 \otimes \ldots \otimes u_n \qquad \text{[Ogden78, Shaw78].}$$

3.2.2.4 Unendliche Verschränkung

Ebenso wie die n-fache Verschränkung einer Menge mit sich selbst, kann auch die Verschränkung von n beliebigen Mengen $U_1, \ldots, U_n \subseteq \Sigma^*$ induktiv definiert werden:

$$\bigotimes_{i=1}^{n} U_i = \begin{cases} \{ \langle\rangle \} & \text{für } n = 0, \\ \left(\bigotimes_{i=1}^{n-1} U_i \right) \otimes U_n & \text{für } n > 0. \end{cases}$$

Anschaulich bedeutet das, daß man aus jeder Menge U_i ein Wort u_i wählt, die Menge aller möglichen Verschränkungen dieser Worte bestimmt und diesen Vorgang für alle möglichen Kombinationen der Worte $u_i \in U_i$ wiederholt.

Versucht man, auf dieselbe Art und Weise *unendlich viele* Mengen U_ω ($\omega \in \Omega$) miteinander zu verschränken, d. h. aus jeder Menge U_ω jeweils ein Wort u_ω zu wählen und diese miteinander zu verschränken, so sind die resultierenden „Worte" (aufgrund der oben erwähnten Beziehung $|w| = |u| + |v|$ für $w \in u \otimes v$) potentiell *unendlich lang* und somit nicht mehr Elemente der Menge Σ^*. Endliche Worte $\in \Sigma^*$ erhält man offensichtlich nur, wenn von den gewählten Worten $u_\omega \in U_\omega$ *fast alle* (d. h. alle bis auf endlich viele) *leer* sind und somit bei der Verschränkung ignoriert werden können. Sofern also alle Mengen U_ω das leere Wort $\langle\rangle$ enthalten, genügt es, jeweils *endlich* viele Worte $u_{\omega_i} \in U_{\omega_i}$ (für paarweise verschiedene Werte $\omega_1, \ldots, \omega_n \in \Omega$) miteinander zu verschränken und diesen Vorgang für alle möglichen Kombinationen von Werten $\omega_1, \ldots, \omega_n$ zu wiederholen. Dies wird durch die folgende Definition formalisiert:[3]

[3] Durch die Notation $\omega_1 \neq \ldots \neq \omega_n \in \Omega$ soll ausgedrückt werden, daß die Werte $\omega_1, \ldots, \omega_n \in \Omega$ *paarweise verschieden* sind, d. h. daß $\omega_i \neq \omega_j$ für $i \neq j$ gilt.

$$\bigotimes_{\omega \in \Omega} U_\omega = \bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} U_{\omega_i}, \quad \text{falls } \forall\, \omega \in \Omega\colon \langle\rangle \in U_\omega.$$

Sofern nur *fast alle* Mengen U_ω das leere Wort enthalten, könnte man zunächst die unendlich vielen Mengen, die das leere Wort enthalten, gemäß dieser Definition miteinander verschränken und anschließend das Resultat mit den endlich vielen „Ausnahmemengen“ verschränken. Sobald jedoch *unendlich viele* Mengen U_ω das leere Wort *nicht* enthalten, würde man bei der Verschränkung aller U_ω nur noch unendlich lange Worte $\notin \Sigma^*$ erhalten, d. h. in diesem Fall muß $\bigotimes_{\omega \in \Omega} U_\omega$ konsequenterweise als *leere Menge* definiert werden.

Da im weiteren Verlauf dieser Arbeit jedoch nur Mengen U_ω zu verschränken sein werden, die entweder *alle* das leere Wort enthalten oder aber alle das leere Wort *nicht* enthalten, kann die Definition von $\bigotimes_{\omega \in \Omega} U_\omega$ auf die folgenden beiden Fälle beschränkt werden:

$$\bigotimes_{\omega \in \Omega} U_\omega = \begin{cases} \displaystyle\bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} U_{\omega_i}, & \text{falls } \forall\, \omega \in \Omega\colon \langle\rangle \in U_\omega, \\ \varnothing, & \text{falls } \exists\, \omega \in \Omega\colon \langle\rangle \notin U_\omega. \end{cases}$$

3.2.3 Vollständige und partielle Worte eines Ausdrucks

Die Semantik eines *regulären Ausdrucks* x wird gewöhnlich als *Sprache* $\mathrm{L}(x)$, d. h. als die Menge aller von diesem Ausdruck *akzeptierten* Worte, definiert [Hopcroft90, Schöning95]. In ähnlicher Weise wird im nachfolgenden Abschnitt 3.3 auch die Semantik von Interaktionsausdrücken bzw. -graphen definiert, wobei ein akzeptiertes Wort hier einer Folge von passierten Aktionen entspricht, die man erhält, wenn man den betrachteten Graphen *vollständig* durchläuft. Daher werden diese Worte auch als *vollständige Worte* des korrespondierenden Ausdrucks bezeichnet.

Durchläuft man einen Graphen nur *teilweise*, d. h. bricht man seine Traversierung vorzeitig ab, so erhält man ein *partielles Wort* des zugehörigen Ausdrucks. Auf den ersten Blick könnte man vermuten, daß jedes partielle Wort eines Ausdrucks bzw. Graphen ein *Präfix* eines vollständigen Worts dieses Ausdrucks darstellt, d. h. daß sich die Menge der partiellen Worte stets mittels Präfixbildung aus der Menge der vollständigen Worte ableiten läßt. Für Graphen, die Sackgassen oder endlose Wege enthalten (vgl. § 2.5), trifft dies jedoch nicht zu: Derartige Graphen können partielle Worte besitzen, die sich *nicht* zu einem vollständigen Wort erweitern lassen.

Da man sich in praktischen Anwendungen von Interaktionsausdrücken primär für die *partiellen* Worte eines Ausdrucks interessiert (vgl. § 4.1.1), könnte man auf den ersten Blick auf die Definition seiner *vollständigen* Worte verzichten. Allerdings benötigt man für eine sinnvolle Definition der partiellen Worte einer sequentiellen Komposition oder Iteration (Sequenz oder Wiederholung) auch die *vollständigen* Worte seiner Teilausdrücke (vgl. § 3.3.1.2 und § 3.3.1.3). Daher ist es erforderlich, für einen Ausdruck x *sowohl* die Menge seiner partiellen Worte (im folgenden mit $\Psi(x)$ bezeichnet) *als auch* die Menge seiner vollständigen Worte (im folgenden mit $\Phi(x)$ bezeichnet)[4] separat und unabhängig voneinander zu definieren.

[4] Als Merkhilfe: **p**si steht für **p**artielle, **ph**i für **v**ollständige Worte.

3.3 Definition von Interaktionsausdrücken

Ein *Interaktionsausdruck* (oder kurz *Ausdruck*) ist:

- ein *elementarer Ausdruck* (vgl. § 3.3.1),
- ein *Multiplikatorausdruck* (vgl. § 3.3.2.1) oder
- ein *Quantorausdruck* (vgl. § 3.3.3).

Die Menge Ξ bezeichne die Menge aller Interaktionsausdrücke.

3.3.1 Elementare Ausdrücke

Ein *elementarer Ausdruck* ist:

- ein *atomarer Ausdruck* (vgl. § 3.3.1.1),
- eine *sequentielle Komposition* (vgl. § 3.3.1.2),
- eine *sequentielle Iteration* (vgl. § 3.3.1.3),
- eine *Disjunktion* (vgl. § 3.3.1.4),
- eine *Option* (vgl. § 3.3.1.5),
- eine *parallele Komposition* (vgl. § 3.3.1.6),
- eine *parallele Iteration* (vgl. § 3.3.1.7),
- eine *Konjunktion* (vgl. § 3.3.1.8) oder
- eine *Synchronisation* (vgl. § 3.3.1.10).

3.3.1.1 Atomare Ausdrücke

Definitionen

Ein *atomarer Ausdruck* ist eine abstrakte Aktion $a = [a_0, a_1, \ldots, a_n] \in \Gamma$ mit dem Namen $a_0 \in \Lambda$ und den (optionalen) Argumenten $a_1, \ldots, a_n \in \Pi \cup \Omega$ (vgl. Abb. 3.1).

$$\boxed{a_0}$$
$$a_1, \ldots, a_n$$

Abbildung 3.1: Atomarer Ausdruck $[a_0, a_1, \ldots, a_n]$

Um diesen Graphen vollständig von links nach rechts zu durchlaufen, muß die Aktion a_0 mit den Argumenten $a_1, \ldots, a_n$ genau einmal passiert werden. Da das Durchlaufen einer Aktion in einem Interaktionsgraphen der Ausführung dieser Aktion in der realen Welt entspricht, ist dies jedoch nur möglich, wenn es sich bei allen Argumenten a_i um konkrete Werte $\in \Omega$ handelt, d. h. wenn a eine *konkrete* Aktion $\in \Sigma$ darstellt. Daher enthält die Menge der vollständigen Worte des Ausdrucks a in diesem Fall genau das Wort $\langle a \rangle$. Andernfalls, d. h. wenn die Aktion a einen oder mehrere formale Parameter als Argumente besitzt, kann der Graph in Abb. 3.1 nicht vollständig durchlaufen werden, d. h. in diesem Fall ist die Menge der vollständigen Worte des Ausdrucks a leer. Dies führt zu folgender Defini-

tion:

$$\Phi(a) = \{\,\langle a\rangle\,\} \cap \Sigma^* = \begin{cases} \{\,\langle a\rangle\,\}, & \text{falls } a \in \Sigma, \\ \varnothing & \text{sonst.} \end{cases}$$

Will man den Graphen in Abb. 3.1 nur teilweise durchlaufen, so muß man die Traversierung entweder vor oder nach dem Durchlaufen der Aktion a abbrechen, wobei letzteres natürlich nur möglich ist, wenn die Aktion a durchlaufen werden kann. Im ersten Fall erhält man ein leeres Wort, im zweiten Fall wieder das Wort $\langle a\rangle$. Zusammengefaßt erhält man daher die folgende Menge von partiellen Worten des Ausdrucks a:

$$\Psi(a) = \{\,\langle\rangle, \langle a\rangle\,\} \cap \Sigma^* = \begin{cases} \{\,\langle\rangle, \langle a\rangle\,\}, & \text{falls } a \in \Sigma, \\ \{\,\langle\rangle\,\} & \text{sonst.} \end{cases}$$

Beispiele

Für den Ausdruck $[\text{Kopie A4, X5}]$ mit $\text{X5} \in \Omega$ (vgl. Abb. 3.2) gilt:

$$\Phi\big([\text{Kopie A4, X5}]\big) = \{\,\langle[\text{Kopie A4, X5}]\rangle\,\} \cap \Sigma^* = \{\,\langle[\text{Kopie A4, X5}]\rangle\,\}$$

und

$$\Psi\big([\text{Kopie A4, X5}]\big) = \{\,\langle\rangle, \langle[\text{Kopie A4, X5}]\rangle\,\} \cap \Sigma^* = \{\,\langle\rangle, \langle[\text{Kopie A4, X5}]\rangle\,\},$$

da es sich bei der Aktion $[\text{Kopie A4, X5}]$ um eine konkrete Aktion $\in \Sigma$ handelt.

Kopie A4
X5

Abbildung 3.2: Atomarer Ausdruck $[\text{Kopie A4, X5}]$

Für den Ausdruck $[\text{Kopie A4}, k]$ mit $k \in \Pi$ (vgl. Abb. 3.3) gilt jedoch:

$$\Phi\big([\text{Kopie A4}, k]\big) = \{\,\langle[\text{Kopie A4}, k]\rangle\,\} \cap \Sigma^* = \varnothing$$

und

$$\Psi\big([\text{Kopie A4}, k]\big) = \{\,\langle\rangle, \langle[\text{Kopie A4}, k]\rangle\,\} \cap \Sigma^* = \{\,\langle\rangle\,\},$$

da es sich bei der Aktion $[\text{Kopie A4}, k]$ um eine abstrakte Aktion $\in \Gamma \setminus \Sigma$ handelt.

Kopie A4
k

Abbildung 3.3: Atomarer Ausdruck $[\text{Kopie A4}, k]$

Anmerkung

Aufgrund der obigen Überlegungen und Definitionen ist die Verwendung formaler Parameter zunächst nicht sehr sinnvoll. Durch die Einführung *konkretisierter* Aktionen und Ausdrücke in § 3.3.3.1 wird

sich dies jedoch ändern, da auf diese Weise formale Parameter im Kontext von *Quantorausdrücken* durch konkrete Werte ersetzt werden können.

Alternativ könnte man die Menge der vollständigen und partiellen Worte eines Ausdrucks abhängig von einer *Parameterbelegungsfunktion* $B: \Pi \to \Omega$ (oft auch als *Umgebung* oder *environment* bezeichnet) definieren, die jedem formalen Parameter $p \in \Pi$ einen konkreten Wert $\omega \in \Omega$ zuordnet. Dies hat jedoch den Nachteil, daß die Belegungsfunktion B stets als zusätzlicher Parameter von Ψ und Φ mitgeführt werden muß. Außerdem müßte man bei diesem Ansatz Ausdrücke mit *global ungebundenen* Parametern verbieten, weil sie keine eigene, kontextunabhängige Semantik besitzen.[5]

3.3.1.2 Sequentielle Komposition

Definitionen

Eine *sequentielle Komposition* (oder Sequenz) ist ein Ausdruck $y - z$ (vgl. Abb. 3.4) mit beliebigen Teilausdrücken y und z, die als *Komponenten* der sequentiellen Komposition bezeichnet werden.

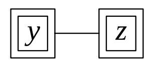


Abbildung 3.4: Sequentielle Komposition $y - z$

Um den Graphen in Abb. 3.4 vollständig zu durchlaufen, muß zunächst der Teilgraph y und anschließend der Teilgraph z vollständig durchlaufen werden, d. h. man erhält ein vollständiges Wort des Ausdrucks $y - z$, indem man ein vollständiges Wort von y mit einem vollständigen Wort von z konkateniert:

$$\Phi(y - z) = \Phi(y)\,\Phi(z).$$

Will man den Graphen nur teilweise durchlaufen, so bricht man die Traversierung entweder innerhalb des Teilgraphen y ab, oder man durchläuft y vollständig und bricht innerhalb des Teilgraphen z ab. Folglich erhält man ein partielles Wort des Ausdrucks $y - z$, indem man entweder ein partielles Wort von y nimmt, oder aber ein *vollständiges* Wort von y mit einem partiellen Wort von z konkateniert:

$$\Psi(y - z) = \Psi(y) \cup \Phi(y)\,\Psi(z).$$

Beispiel

Für den Ausdruck 10 Pf – Kopie A4 (vgl. Abb. 3.5) ergibt sich die Menge der vollständigen bzw. partiellen Worte wie folgt:

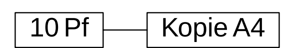


Abbildung 3.5: Sequentielle Komposition 10 Pf – Kopie A4

[5] Ein formaler Parameter eines Ausdrucks ist global ungebunden, wenn es keinen umgebenden Quantor mit diesem Parameter gibt (vgl. § 3.3.3).
Zweifellos sind Ausdrücke mit global ungebundenen Parametern nicht besonders sinnvoll; es vereinfacht jedoch die Implementierung des Formalismus, wenn man sie nicht grundsätzlich verbietet.

Φ(10 Pf – Kopie A4) = Φ(10 Pf) Φ(Kopie A4) = { ⟨10 Pf⟩ } { ⟨Kopie A4⟩ } = { ⟨10 Pf⟩ ⟨Kopie A4⟩ } = { ⟨10 Pf, Kopie A4⟩ };

Ψ(10 Pf – Kopie A4) = Ψ(10 Pf) ∪ Φ(10 Pf) Ψ(Kopie A4) =
{ ⟨⟩, ⟨10 Pf⟩ } ∪ { ⟨10 Pf⟩ } { ⟨⟩, ⟨Kopie A4⟩ } = { ⟨⟩, ⟨10 Pf⟩ } ∪ { ⟨10 Pf⟩ ⟨⟩, ⟨10 Pf⟩ ⟨Kopie A4⟩ } =
{ ⟨⟩, ⟨10 Pf⟩ } ∪ { ⟨10 Pf⟩, ⟨10 Pf, Kopie A4⟩ } = { ⟨⟩, ⟨10 Pf⟩, ⟨10 Pf, Kopie A4⟩ }.

3.3.1.3 Sequentielle Iteration

Definitionen

Eine *sequentielle Iteration* (oder Wiederholung) ist ein Ausdruck $\ominus\, y$ (vgl. Abb. 3.6) mit einem beliebigen Teilausdruck y, der als *Rumpf* (engl. body) der Iteration bezeichnet wird.

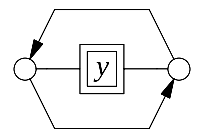


Abbildung 3.6: Sequentielle Iteration $\ominus\, y$

Um den Graphen in Abb. 3.6 vollständig zu durchlaufen, muß der Teilgraph y beliebig oft (evtl. auch keinmal) vollständig durchlaufen werden, d. h. man erhält ein vollständiges Wort des Ausdrucks $\ominus\, y$, indem man beliebig viele vollständige Worte von y konkateniert:

$\Phi(\ominus\, y) = \Phi(y)^*$.

Will man den Graphen nur teilweise durchlaufen, so bricht man die Traversierung von y beim *letzten* Iterationsschritt vorzeitig ab, d. h. man erhält ein partielles Wort des Ausdrucks $\ominus\, y$, indem man beliebig viele *vollständige* Worte mit einem partiellen Wort von y konkateniert:

$\Psi(\ominus\, y) = \Phi(y)^*\, \Psi(y)$.

Beispiel

Der Ausdruck $\ominus$ (10 Pf – Kopie A4) (vgl. Abb. 3.7) besitzt die folgenden vollständigen bzw. partiellen Worte:

Φ($\ominus$ (10 Pf – Kopie A4)) = Φ(10 Pf – Kopie A4)* = { ⟨10 Pf, Kopie A4⟩ }* =
{ ⟨⟩, ⟨10 Pf, Kopie A4⟩, ⟨10 Pf, Kopie A4, 10 Pf, Kopie A4⟩, . . . };

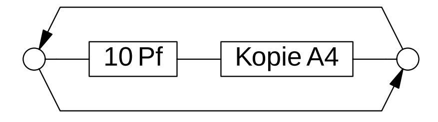


Abbildung 3.7: Sequentielle Iteration $\ominus$ (10 Pf – Kopie A4)

$\Psi(\ominus\,(\text{10 Pf} - \text{Kopie A4})) = \Phi(\text{10 Pf} - \text{Kopie A4})^{*}\,\Psi(\text{10 Pf} - \text{Kopie A4}) =$
$\{\,\langle\text{10 Pf, Kopie A4}\rangle\,\}^{*}\,\{\,\langle\rangle, \langle\text{10 Pf}\rangle, \langle\text{10 Pf, Kopie A4}\rangle\,\} =$
$\{\,\langle\rangle, \langle\text{10 Pf}\rangle, \langle\text{10 Pf, Kopie A4}\rangle, \langle\text{10 Pf, Kopie A4, 10 Pf}\rangle, \langle\text{10 Pf, Kopie A4, 10 Pf, Kopie A4}\rangle, \ldots\,\}.$

3.3.1.4 Disjunktion

Definitionen

Eine *Disjunktion* (oder Entweder-oder-Verzweigung) ist ein Ausdruck $y \bigcirc z$ (vgl. Abb. 3.8) mit beliebigen Teilausdrücken y und z, die als *Alternativen* oder *Zweige* der Disjunktion bezeichnet werden.

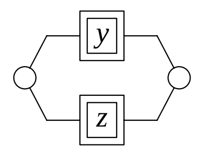


Abbildung 3.8: Disjunktion $y \bigcirc z$

Um den Graphen in Abb. 3.8 vollständig bzw. teilweise zu durchlaufen, muß *einer* der beiden Teilgraphen y oder z vollständig bzw. teilweise durchlaufen werden, d. h. der Ausdruck $y \bigcirc z$ besitzt die folgenden vollständigen bzw. partiellen Worte:

$\Phi(y \bigcirc z) = \Phi(y) \cup \Phi(z),$

$\Psi(y \bigcirc z) = \Psi(y) \cup \Psi(z).$

Beispiel

Für den Ausdruck $\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})$ (vgl. Abb. 3.9) gilt:

$\Phi(\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})) = \Phi(\text{2 DM}) \cup \Phi(\text{1 DM} - \text{1 DM}) = \{\,\langle\text{2 DM}\rangle\,\} \cup \{\,\langle\text{1 DM, 1 DM}\rangle\,\} =$
$\{\,\langle\text{2 DM}\rangle, \langle\text{1 DM, 1 DM}\rangle\,\};$

$\Psi(\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})) = \Psi(\text{2 DM}) \cup \Psi(\text{1 DM} - \text{1 DM}) =$
$\{\,\langle\rangle, \langle\text{2 DM}\rangle\,\} \cup \{\,\langle\rangle, \langle\text{1 DM}\rangle, \langle\text{1 DM, 1 DM}\rangle\,\} = \{\,\langle\rangle, \langle\text{2 DM}\rangle, \langle\text{1 DM}\rangle, \langle\text{1 DM, 1 DM}\rangle\,\}.$

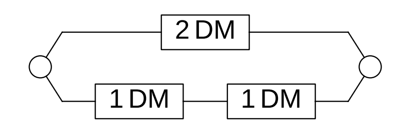


Abbildung 3.9: Disjunktion $\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})$

3.3.1.5 Option

Definitionen

Eine *Option* (oder Eventuell-Verzweigung) ist ein Ausdruck $\circlearrowright y$ (vgl. Abb. 3.10) mit einem beliebigen Teilausdruck y, der als *Rumpf* der Option bezeichnet wird.

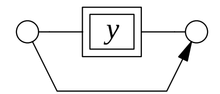


Abbildung 3.10: Option $\circlearrowright y$

Der Graph in Abb. 3.10 wird durchlaufen, indem der Teilgraph y entweder durchlaufen oder umgangen wird. Daher akzeptiert der Ausdruck $\circlearrowright y$ dieselben vollständigen und partiellen Worte wie der Teilausdruck y und zusätzlich das leere Wort. Da die Menge $\Psi(y)$ das leere Wort bereits enthält (vgl. § 3.4.4), muß es nicht mehr explizit hinzugefügt werden:

$$\Phi(\circlearrowright y) = \Phi(y) \cup \{ \langle\rangle \},$$

$$\Psi(\circlearrowright y) = \Psi(y).$$

Beispiel

Für den Ausdruck $a - \circlearrowright(b - c)$ (vgl. Abb. 3.11) gilt:

$$\Phi(a - \circlearrowright(b - c)) = \Phi(a)\,\Phi(\circlearrowright(b - c)) = \Phi(a)\big(\Phi(b - c) \cup \{ \langle\rangle \}\big) = \{ \langle a\rangle \}\big(\{ \langle b, c\rangle \} \cup \{ \langle\rangle \}\big) = \{ \langle a\rangle \}\, \{ \langle\rangle, \langle b, c\rangle \} = \{ \langle a\rangle, \langle a, b, c\rangle \};$$

$$\Psi(a - \circlearrowright(b - c)) = \Psi(a) \cup \Phi(a)\,\Psi(\circlearrowright(b - c)) = \Psi(a) \cup \Phi(a)\,\Psi(b - c) = \{ \langle\rangle, \langle a\rangle \} \cup \{ \langle a\rangle \}\, \{ \langle\rangle, \langle b\rangle, \langle b, c\rangle \} = \{ \langle\rangle, \langle a\rangle \} \cup \{ \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle \} = \{ \langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle \}.$$

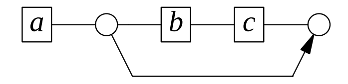


Abbildung 3.11: Ausdruck $a - \circlearrowright(b - c)$ mit Option

3.3.1.6 Parallele Komposition

Definitionen

Eine *parallele Komposition* (oder Sowohl-als-auch-Verzweigung) ist ein Ausdruck $y \circledcirc z$ (vgl. Abb. 3.12) mit beliebigen Teilausdrücken y und z, die als *Komponenten* oder *Zweige* der parallelen Komposition bezeichnet werden.

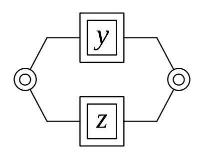


Abbildung 3.12: Parallele Komposition $y \circledcirc z$

Um den Graphen in Abb. 3.12 vollständig zu durchlaufen, müssen die beiden Teilgraphen y und z unabhängig voneinander vollständig durchlaufen werden. Daher erhält man ein vollständiges Wort des Ausdrucks $y \circledcirc z$, indem man ein vollständiges Wort von y mit einem vollständigen Wort von z verschränkt:

$$\Phi(y \circledcirc z) = \Phi(y) \otimes \Phi(z).$$

Will man den Graphen nur teilweise durchlaufen, so bricht man die Traversierung der Teilgraphen y und z jeweils an einer beliebigen Stelle ab, d. h. man erhält ein partielles Wort des Ausdrucks $y \circledcirc z$, indem man ein partielles Wort von y mit einem partiellen Wort von z verschränkt:

$$\Psi(y \circledcirc z) = \Psi(y) \otimes \Psi(z).$$

Beispiel

Der Ausdruck $(2\,\mathsf{DM} \bigcirc (1\,\mathsf{DM} - 1\,\mathsf{DM})) \circledcirc 1\,\mathsf{DM}$ (vgl. Abb. 3.13) besitzt die folgenden vollständigen bzw. partiellen Worte:[6]

$$\begin{aligned}
&\Phi((2\,\mathsf{DM} \bigcirc (1\,\mathsf{DM} - 1\,\mathsf{DM})) \circledcirc 1\,\mathsf{DM}) = \Phi(2\,\mathsf{DM} \bigcirc (1\,\mathsf{DM} - 1\,\mathsf{DM})) \otimes \Phi(1\,\mathsf{DM}) = \\
&\{\,\langle 2\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\} \otimes \{\,\langle 1\,\mathsf{DM}\rangle\,\} = \langle 2\,\mathsf{DM}\rangle \otimes \langle 1\,\mathsf{DM}\rangle \cup \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle \otimes \langle 1\,\mathsf{DM}\rangle = \\
&\{\,\langle 2\,\mathsf{DM}, 1\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 2\,\mathsf{DM}\rangle\,\} \cup \{\,\langle 1\,\mathsf{DM}, 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\} = \\
&\{\,\langle 2\,\mathsf{DM}, 1\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 2\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\};
\end{aligned}$$

$$\begin{aligned}
&\Psi((2\,\mathsf{DM} \bigcirc (1\,\mathsf{DM} - 1\,\mathsf{DM})) \circledcirc 1\,\mathsf{DM}) = \Psi(2\,\mathsf{DM} \bigcirc (1\,\mathsf{DM} - 1\,\mathsf{DM})) \otimes \Psi(1\,\mathsf{DM}) = \\
&\{\,\langle\rangle, \langle 2\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\} \otimes \{\,\langle\rangle, \langle 1\,\mathsf{DM}\rangle\,\} = \\
&\langle\rangle \otimes \langle\rangle \cup \langle\rangle \otimes \langle 1\,\mathsf{DM}\rangle \cup \langle 2\,\mathsf{DM}\rangle \otimes \langle\rangle \cup \langle 2\,\mathsf{DM}\rangle \otimes \langle 1\,\mathsf{DM}\rangle \cup \\
&\langle 1\,\mathsf{DM}\rangle \otimes \langle\rangle \cup \langle 1\,\mathsf{DM}\rangle \otimes \langle 1\,\mathsf{DM}\rangle \cup \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle \otimes \langle\rangle \cup \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle \otimes \langle 1\,\mathsf{DM}\rangle = \\
&\{\,\langle\rangle\,\} \cup \{\,\langle 1\,\mathsf{DM}\rangle\,\} \cup \{\,\langle 2\,\mathsf{DM}\rangle\,\} \cup \{\,\langle 2\,\mathsf{DM}, 1\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 2\,\mathsf{DM}\rangle\,\} \cup \\
&\{\,\langle 1\,\mathsf{DM}\rangle\,\} \cup \{\,\langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\} \cup \{\,\langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\} \cup \{\,\langle 1\,\mathsf{DM}, 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\} = \\
&\{\,\langle\rangle, \langle 1\,\mathsf{DM}\rangle, \langle 2\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 2\,\mathsf{DM}\rangle, \langle 2\,\mathsf{DM}, 1\,\mathsf{DM}\rangle, \langle 1\,\mathsf{DM}, 1\,\mathsf{DM}, 1\,\mathsf{DM}\rangle\,\}.
\end{aligned}$$

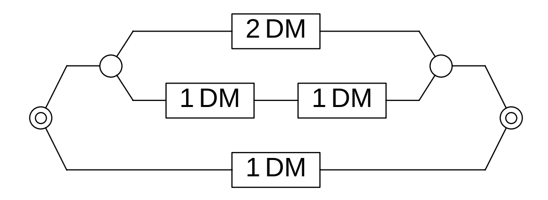


Abbildung 3.13: Parallele Komposition $(2\,\mathsf{DM} \bigcirc (1\,\mathsf{DM} - 1\,\mathsf{DM})) \circledcirc 1\,\mathsf{DM}$

[6] Der Verschränkungsoperator $\otimes$ soll einen höheren Vorrang besitzen als der Vereinigungsoperator $\cup$ und der Durchschnittsoperator $\cap$.

3.3.1.7 Parallele Iteration

Definitionen

Eine *parallele Iteration* (oder Beliebig-oft-Verzweigung) ist ein Ausdruck $\circledcirc\, y$ (vgl. Abb. 3.14) mit einem beliebigen Teilausdruck y, der als *Rumpf* der Iteration bezeichnet wird.

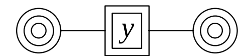


Abbildung 3.14: Parallele Iteration $\circledcirc\, y$

Um den Graphen in Abb. 3.14 vollständig zu durchlaufen, müssen beliebig viele Ausprägungen des Teilgraphen y unabhängig voneinander vollständig durchlaufen werden, d. h. man erhält ein vollständiges Wort des Ausdrucks $\circledcirc\, y$, indem man beliebig viele vollständige Worte von y miteinander verschränkt:

$$\Phi(\circledcirc\, y) = \Phi(y)\#.$$

Ein partielles Wort erhält man entsprechend, indem man beliebig viele partielle Worte von y miteinander verschränkt:

$$\Psi(\circledcirc\, y) = \Psi(y)\#.$$

Beispiel

Für die Menge $C = \Phi(\text{10 Pf} - \text{Kopie A4}) = \{\, \langle \text{10 Pf, Kopie A4} \rangle \,\}$ gilt:

$$\overset{0}{\bigotimes} C = \{\, \langle\rangle \,\},$$

$$\overset{1}{\bigotimes} C = C = \{\, \langle \text{10 Pf, Kopie A4} \rangle \,\},$$

$$\overset{2}{\bigotimes} C = C \otimes C = \{\, \langle \text{10 Pf, Kopie A4} \rangle \,\} \otimes \{\, \langle \text{10 Pf, Kopie A4} \rangle \,\} =$$
$$\{\, \langle \text{10 Pf, 10 Pf, Kopie A4, Kopie A4} \rangle, \langle \text{10 Pf, Kopie A4, 10 Pf, Kopie A4} \rangle \,\},$$

$$\overset{3}{\bigotimes} C = \left(\overset{2}{\bigotimes} C \right) \otimes C =$$
$$\{\, \langle \text{10 Pf, 10 Pf, Kopie A4, Kopie A4} \rangle, \langle \text{10 Pf, Kopie A4, 10 Pf, Kopie A4} \rangle \,\} \otimes \{\, \langle \text{10 Pf, Kopie A4} \rangle \,\} =$$
$$\{\, \langle \text{10 Pf, 10 Pf, 10 Pf, Kopie A4, Kopie A4, Kopie A4} \rangle,$$
$$\langle \text{10 Pf, 10 Pf, Kopie A4, 10 Pf, Kopie A4, Kopie A4} \rangle,$$
$$\langle \text{10 Pf, 10 Pf, Kopie A4, Kopie A4, 10 Pf, Kopie A4} \rangle,$$
$$\langle \text{10 Pf, Kopie A4, 10 Pf, 10 Pf, Kopie A4, Kopie A4} \rangle,$$
$$\langle \text{10 Pf, Kopie A4, 10 Pf, Kopie A4, 10 Pf, Kopie A4} \rangle \,\},$$

usw.

Daher besitzt der Ausdruck $\circledcirc\, (\text{10 Pf} - \text{Kopie A4})$ (vgl. Abb. 3.15) die folgenden vollständigen Worte:

$$\Phi(\circledcirc\, (\text{10 Pf} - \text{Kopie A4})) = \Phi(\text{10 Pf} - \text{Kopie A4})\# = \bigcup_{n=0}^{\infty} \overset{n}{\bigotimes} \Phi(\text{10 Pf} - \text{Kopie A4}) =$$
$$\{\, \langle\rangle, \langle \text{10 Pf, Kopie A4} \rangle, \langle \text{10 Pf, 10 Pf, Kopie A4, Kopie A4} \rangle, \langle \text{10 Pf, Kopie A4, 10 Pf, Kopie A4} \rangle,$$

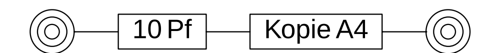


Abbildung 3.15: Parallele Iteration ◎ (10 Pf – Kopie A4)

⟨10 Pf, 10 Pf, 10 Pf, Kopie A4, Kopie A4, Kopie A4⟩,
⟨10 Pf, 10 Pf, Kopie A4, 10 Pf, Kopie A4, Kopie A4⟩,
⟨10 Pf, 10 Pf, Kopie A4, Kopie A4, 10 Pf, Kopie A4⟩,
⟨10 Pf, Kopie A4, 10 Pf, 10 Pf, Kopie A4, Kopie A4⟩,
⟨10 Pf, Kopie A4, 10 Pf, Kopie A4, 10 Pf, Kopie A4⟩, ... }.

In ähnlicher Weise ergibt sich die Menge seiner partiellen Worte als:

$$\Psi(◎\ (\text{10 Pf} - \text{Kopie A4})) = \Psi(\text{10 Pf} - \text{Kopie A4})\# = \bigcup_{n=0}^{\infty} \bigotimes^{n} \Psi(\text{10 Pf} - \text{Kopie A4}) =$$

{ ⟨⟩, ⟨10 Pf⟩, ⟨10 Pf, Kopie A4⟩,
⟨10 Pf, 10 Pf⟩, ⟨10 Pf, 10 Pf, Kopie A4⟩, ⟨10 Pf, 10 Pf, Kopie A4, Kopie A4⟩,
⟨10 Pf, Kopie A4, 10 Pf⟩, ⟨10 Pf, Kopie A4, 10 Pf, Kopie A4⟩, ... }.

3.3.1.8 Konjunktion

Definitionen

Eine *Konjunktion* (oder *strikte* Kopplung) ist ein Ausdruck $y \bullet z$ (vgl. Abb. 3.16) mit beliebigen Teilausdrücken y und z, die als *Zweige* der Konjunktion bezeichnet werden.

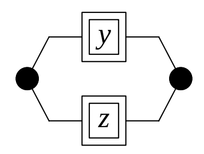


Abbildung 3.16: Konjunktion $y \bullet z$

Der Graph in Abb. 3.16 wird durchlaufen, indem die beiden Teilgraphen y und z *strikt synchronisiert* durchlaufen werden, d. h. eine Aktion darf nur passiert werden, wenn sie in beiden Teilgraphen *gleichzeitig* durchlaufen werden kann. Folglich erlaubt der Ausdruck $y \bullet z$ nur Aktionsfolgen, die *sowohl* in y *als auch* in z zulässig sind. Dies führt zu folgenden Definitionen:

$$\Phi(y \bullet z) = \Phi(y) \cap \Phi(z),$$

$$\Psi(y \bullet z) = \Psi(y) \cap \Psi(z).$$

Anmerkung

Da für praktische Anwendungen in aller Regel die in § 3.3.1.10 definierte *Synchronisation* gegenüber der Konjunktion bevorzugt wird, wurde letztere in Kapitel 2 nicht explizit eingeführt. Allerdings stellt die Konjunktion einerseits ein nützliches Hilfsmittel zur Definition der Synchronisation dar (vgl. § 3.3.1.10), andererseits gehört sie als duales Pendant zur Disjunktion auch aus Gründen der konzeptionellen Vollständigkeit zur Menge der „Interaktionsoperatoren“.

3.3.1.9 Alphabet eines Ausdrucks

Bevor im nachfolgenden Abschnitt 3.3.1.10 der Synchronisationsoperator ⭘ als letzter verbleibender *elementarer Operator* von Interaktionsausdrücken eingeführt werden kann, muß zunächst das *Alphabet* $\alpha(x)$ eines Ausdrucks x, d. h. die *Menge aller Aktionen* von x, definiert werden:

$$\alpha(x) = \begin{cases} \{\, a \,\} & \text{für einen atomaren Ausdruck } x \equiv a \quad \text{mit } a \in \Gamma, \\ \alpha(y) & \text{für einen unären Ausdruck } x \equiv \text{◍}\, y \quad \text{mit ◍} \in \{\, \text{◡}, \ominus, ⊚ \,\}, \\ \alpha(y) \cup \alpha(z) & \text{für einen binären Ausdruck } x \equiv y \,\text{◍}\, z \quad \text{mit ◍} \in \{\, -, \bigcirc, ◎, ⭘, \bullet \,\}. \end{cases}$$

Anmerkung: Da das Gleichheitszeichen zwischen Ausdrücken später zur Notation der *semantischen* Gleichheit (oder Äquivalenz) verwendet wird (vgl. § 3.4.1.1), wird die *syntaktische* Gleichheit (oder Übereinstimmung) von Ausdrücken durch das Zeichen ≡ notiert.

3.3.1.10 Synchronisation

Definitionen

Eine *Synchronisation* (oder [lose] Kopplung) ist ein Ausdruck y ⭘ z (vgl. Abb. 3.17) mit beliebigen Teilausdrücken y und z, die als *Zweige* der Synchronisation bezeichnet werden.

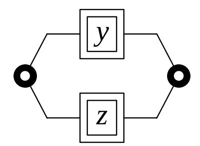


Abbildung 3.17: Synchronisation y ⭘ z

Der Graph in Abb. 3.17 wird durchlaufen, indem die beiden Teilgraphen y und z *teilweise synchronisiert* durchlaufen werden: Aktionen, die nur in *einem* Zweig vorkommen, können − wie bei der parallelen Komposition − *unabhängig* vom anderen Zweig durchlaufen werden, während Aktionen, die *beiden* Zweigen gemeinsam sind, − wie bei der Konjunktion − *gleichzeitig* durchlaufen werden müssen. Um diese Regeln formal zu fassen, wird eine Synchronisation zunächst in eine äquivalente Konjunktion transformiert.

Transformation in eine Konjunktion

Bezeichnet man mit $y_1, \ldots, y_m$ (bzw. $z_1, \ldots, z_n$) die Aktionen, die nur im Teilausdruck y (bzw. z), nicht jedoch im Teilausdruck z (bzw. y) vorkommen, so kann der Ausdruck y ⭘ z schrittweise wie folgt umgeformt werden, ohne seine ursprüngliche Bedeutung zu verändern:

1. Zunächst wird der obere Zweig y durch eine parallele Komposition $y \circledcirc \ominus(z_1 \bigcirc \ldots \bigcirc z_n)$ ersetzt (vgl. Abb. 3.18). Dadurch sind die Aktionen $z_1, \ldots, z_n$ jetzt beiden Zweigen der Synchronisation gemeinsam, d. h. sie dürfen nur noch durchlaufen werden, wenn sie in *beiden* Zweigen *gleichzeitig* passiert werden können.
   Da die sequentielle Iteration $\ominus(z_1 \bigcirc \ldots \bigcirc z_n)$ jedoch *unabhängig* vom Teilgraphen y durchlaufen wird, können die Aktionen $z_1, \ldots, z_n$ im oberen Zweig der Synchronisation *jederzeit* passiert werden, d. h. im Gesamtausdruck dürfen sie nach wie vor genau dann durchlaufen werden, wenn sie

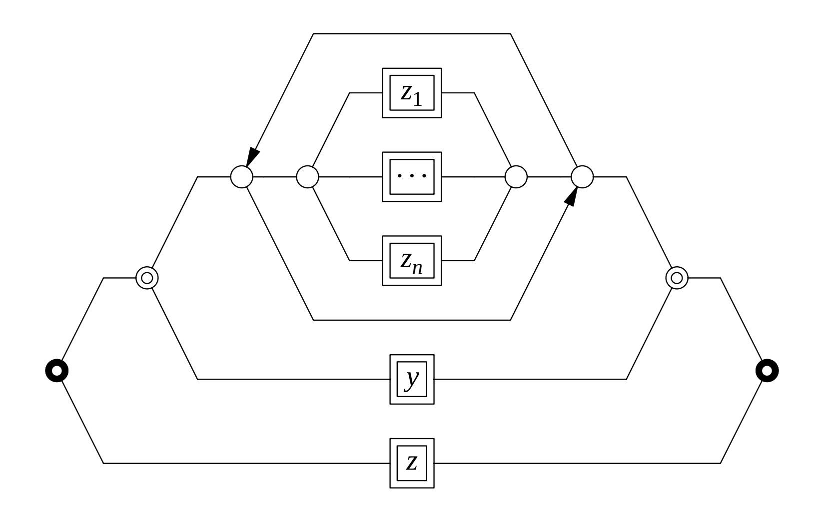


Abbildung 3.18: Transformation der Synchronisation *y* ⚬ *z* (Schritt 1)

im unteren Zweig *z* durchlaufen werden können.
Alle übrigen Aktionen des Ausdrucks sind von dieser Änderung nicht betroffen.

2. Analog zu Schritt 1, kann auch der untere Zweig *z* der Synchronisation durch eine parallele Komposition $z \circledcirc \ominus(y_1 \bigcirc \ldots \bigcirc y_m)$ ersetzt werden (vgl. Abb. 3.19), ohne die Bedeutung des Ausdrucks zu verändern.

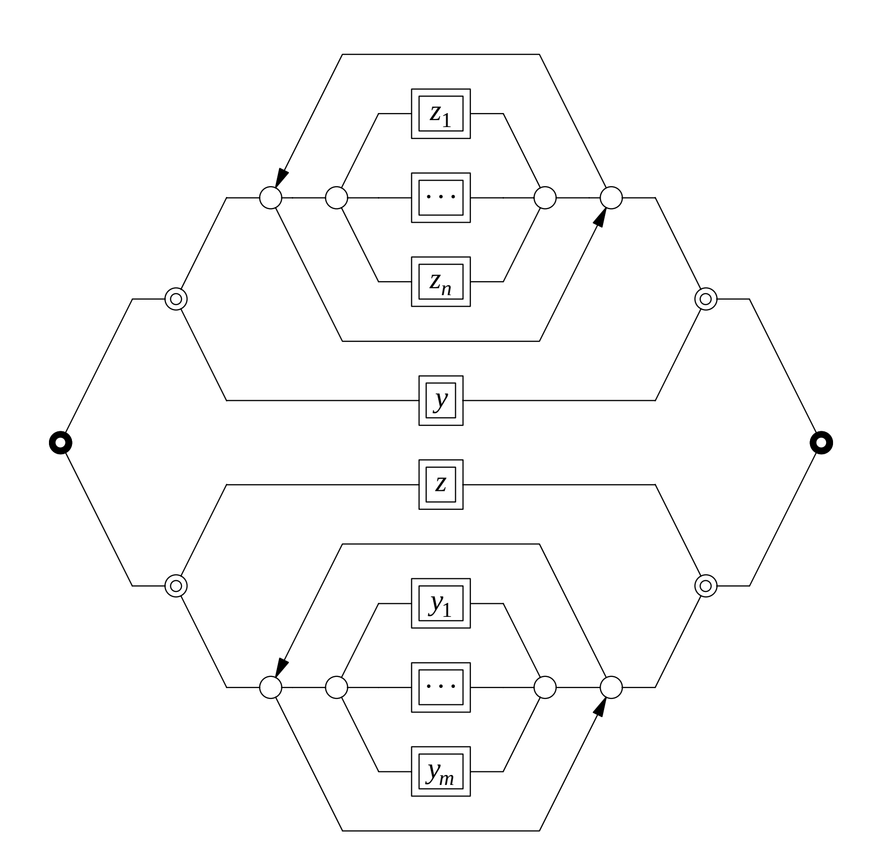


Abbildung 3.19: Transformation der Synchronisation *y* ⚬ *z* (Schritt 2)

3. Da nun beide Zweige der Synchronisation *dasselbe Alphabet* besitzen, können *sämtliche* Aktionen nur noch durchlaufen werden, wenn sie in beiden Zweigen gleichzeitig passiert werden können, d. h. die Synchronisation in Abb. 3.19 ist äquivalent zu einer *Konjunktion* mit denselben Zweigen.

Zusammengefaßt erhält man daher die folgende *Äquivalenztransformation*, die es erlaubt, eine Synchronisation auf bereits definierte Operatoren zurückzuführen:

$$y \mathbin{\boldsymbol{\circ}} z = \big(y \circledcirc \ominus\big(z_1 \bigcirc \ldots \bigcirc z_n\big)\big) \bullet \big(z \circledcirc \ominus\big(y_1 \bigcirc \ldots \bigcirc y_m\big)\big).$$

Konsequenzen der Transformation

Bezeichnet man mit $\kappa_x(y)$ bzw. $\kappa_x(z)$ (**k**appa wie **K**omplement) die Menge aller Aktionen des Ausdrucks $x \equiv y \mathbin{\boldsymbol{\circ}} z$, die *nicht* im Teilausdruck y bzw. z vorkommen, d. h.

$$\kappa_x(y) = \alpha(x) \setminus \alpha(y) = (\alpha(y) \cup \alpha(z)) \setminus \alpha(y) = \alpha(z) \setminus \alpha(y) = \{\, z_1, \ldots, z_n \,\},$$

$$\kappa_x(z) = \alpha(x) \setminus \alpha(z) = (\alpha(y) \cup \alpha(z)) \setminus \alpha(z) = \alpha(y) \setminus \alpha(z) = \{\, y_1, \ldots, y_m \,\},$$

so gilt:

$$\Phi\big(\ominus\big(y_1 \bigcirc \ldots \bigcirc y_m\big)\big) = \Phi\big(y_1 \bigcirc \ldots \bigcirc y_m\big)^* = \big(\Phi\big(y_1\big) \cup \ldots \cup \Phi\big(y_n\big)\big)^* =$$
$$\{\, \langle y_1 \rangle, \ldots, \langle y_m \rangle \,\}^* = \{\, y_1, \ldots, y_m \,\}^* = \kappa_x(z)^*,$$

und analog:

$$\Phi\big(\ominus\big(z_1 \bigcirc \ldots \bigcirc z_n\big)\big) = \ldots = \kappa_x(y)^*.$$

Daher ergibt sich die Menge der vollständigen Worte des Ausdrucks $y \mathbin{\boldsymbol{\circ}} z$ wie folgt:

$$\begin{aligned}
\Phi(y \mathbin{\boldsymbol{\circ}} z) &= \Phi\big(\big(y \circledcirc \ominus\big(z_1 \bigcirc \ldots \bigcirc z_n\big)\big) \bullet \big(z \circledcirc \ominus\big(y_1 \bigcirc \ldots \bigcirc y_m\big)\big)\big) = \\
&= \Phi\big(y \circledcirc \ominus\big(z_1 \bigcirc \ldots \bigcirc z_n\big)\big) \cap \Phi\big(z \circledcirc \ominus\big(y_1 \bigcirc \ldots \bigcirc y_m\big)\big) = \\
&= \Phi(y) \otimes \Phi\big(\ominus\big(z_1 \bigcirc \ldots \bigcirc z_n\big)\big) \cap \Phi(z) \otimes \Phi\big(\ominus\big(y_1 \bigcirc \ldots \bigcirc y_m\big)\big) = \\
&= \Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^*,
\end{aligned}$$

und vollkommen analog erhält man die Menge der partiellen Worte:

$$\Psi(y \mathbin{\boldsymbol{\circ}} z) = \Psi(y) \otimes \kappa_x(y)^* \cap \Psi(z) \otimes \kappa_x(z)^*.$$

Anmerkungen

Die soeben entwickelten Formeln können auch wie folgt interpretiert werden:

Ein Wort $w \in \Phi(y) \otimes \kappa_x(y)^*$ entsteht durch Verschränken eines Wortes $u \in \Phi(y)$ mit einem Wort $v' \in \kappa_x(y)^*$, d. h. indem man in ein Wort $u \in \Phi(y)$ an beliebigen Stellen Aktionen $a \in \kappa_x(y)$ einfügt. Ebenso erhält man ein Wort $w \in \Phi(z) \otimes \kappa_x(z)$, indem man in ein Wort $v \in \Phi(z)$ an beliebigen Stellen Aktionen $a \in \kappa_x(z)$ einfügt. Anschaulich entspricht dies der Tatsache, daß der Ausdruck y bzw. z Aktionen, über die er *keine Aussage macht*, jederzeit zuläßt (vgl. § 2.3.2.2).

Dies bedeutet umgekehrt, daß man ein Wort $u \in \Phi(y)$ erhält, indem man aus einem Wort $w \in \Phi(y) \otimes \kappa_x(y)^*$ alle Aktionen $a \in \kappa_x(y)$ entfernt. Entsprechend erhält man ein Wort $v \in \Phi(z)$, wenn man aus einem Wort $w \in \Phi(z) \otimes \kappa_x(z)^*$ alle Aktionen $a \in \kappa_x(z)$ entfernt. Somit enthält $\Phi(y \mathbin{\boldsymbol{\circ}} z)$ genau die Worte w, bei denen man durch Entfernen der Aktionen $y_1, \ldots, y_m$ bzw. $z_1, \ldots, z_n$ ein Wort $u \in \Phi(z)$ bzw. $v \in \Phi(y)$ erhält.

Diese Überlegungen gelten in gleicher Weise auch für die Menge $\Psi(y \mathbin{\boldsymbol{\circ}} z)$ der partiellen Worte.

Eine weitere Anmerkung betrifft Ausdrücke mit *unendlichen* Alphabeten, wie sie z. B. beim Einsatz von Quantoren auftreten (vgl. § 3.3.3): Wenn eine der Mengen $\kappa_x(y)$ oder $\kappa_x(z)$ unendlich ist, lassen sich die obigen Transformationsschritte nicht ohne weiteres durchführen, da Disjunktionen mit unend-

lich vielen Zweigen nicht definiert sind. Die oben hergeleiteten Formeln

$$\Phi(y \mathbin{\boldsymbol{\circ}} z) = \Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^*$$

und

$$\Psi(y \mathbin{\boldsymbol{\circ}} z) = \Psi(y) \otimes \kappa_x(y)^* \cap \Psi(z) \otimes \kappa_x(z)^*$$

sind aber nichtsdestotrotz wohldefiniert und können daher − unabhängig von ihrer anschaulich motivierten Herleitung − als *allgemeingültige Definition* der Synchronisation $y \mathbin{\boldsymbol{\circ}} z$ verwendet werden.

Beispiel

Für die Ausdrücke $y \equiv \circlearrowleft(\text{10 Pf} - \text{Kopie A4})$ und $z \equiv \circlearrowleft\big(\text{Kopie A4} \bigcirc (\text{Öffnen A4} - \text{Schließen A4})\big)$ gilt:

$$\Phi(y) = \{\, \langle\rangle, \langle \text{10 Pf, Kopie A4} \rangle, \langle \text{10 Pf, Kopie A4, 10 Pf, Kopie A4} \rangle, \ldots \},$$

$$\alpha(y) = \{\, \text{10 Pf, Kopie A4} \,\};$$

$$\Phi(z) = \big\{\langle\rangle, \langle \text{Kopie A4} \rangle, \langle \text{Öffnen A4, Schließen A4} \rangle, \langle \text{Kopie A4, Öffnen A4, Schließen A4} \rangle, \\ \langle \text{Öffnen A4, Schließen A4, Kopie A4} \rangle, \ldots\big\},$$

$$\alpha(z) = \big\{\, \text{Kopie A4, Öffnen A4, Schließen A4} \,\big\}.$$

Somit gilt für die Alphabet-Komplement-Mengen $\kappa_x(y)$ und $\kappa_x(z)$ des in Abb. 3.20 dargestellten Ausdrucks $x \equiv y \mathbin{\boldsymbol{\circ}} z$:

$$\kappa_x(y) = \alpha(z) \setminus \alpha(y) = \big\{\, \text{Öffnen A4, Schließen A4} \,\big\},$$

$$\kappa_x(z) = \alpha(y) \setminus \alpha(z) = \{\, \text{10 Pf} \,\}.$$

Weiterhin gilt, wenn man mit *ÖS* jeweils eine beliebige Folge der Aktionen Öffnen A4 und Schließen A4 und mit *ZZ* jeweils eine beliebig lange 10 Pf-Folge bezeichnet:

$$\Phi(y) \otimes \kappa_x(y)^* = \\ \{\, \langle\rangle, \langle \text{10 Pf, Kopie A4} \rangle, \langle \text{10 Pf, Kopie A4, 10 Pf, Kopie A4} \rangle, \ldots \} \otimes \big\{\, \text{Öffnen A4, Schließen A4} \,\big\}^* = \\ \big\{\, \langle ÖS \rangle, \langle ÖS, \text{10 Pf}, ÖS, \text{Kopie A4}, ÖS \rangle, \\ \langle ÖS, \text{10 Pf}, ÖS, \text{Kopie A4}, ÖS, \text{10 Pf}, ÖS, \text{Kopie A4}, ÖS \rangle, \ldots \big\},$$

$$\Phi(z) \otimes \kappa_x(z)^* = \\ \big\{\, \langle\rangle, \langle \text{Kopie A4} \rangle, \langle \text{Öffnen A4, Schließen A4} \rangle, \langle \text{Kopie A4, Öffnen A4, Schließen A4} \rangle, \\ \langle \text{Öffnen A4, Schließen A4, Kopie A4} \rangle, \ldots \big\} \otimes \{\, \text{10 Pf} \,\}^* = \\ \big\{\, \langle ZZ \rangle, \langle ZZ, \text{Kopie A4}, ZZ \rangle, \langle ZZ, \text{Öffnen A4}, ZZ, \text{Schließen A4}, ZZ \rangle,$$

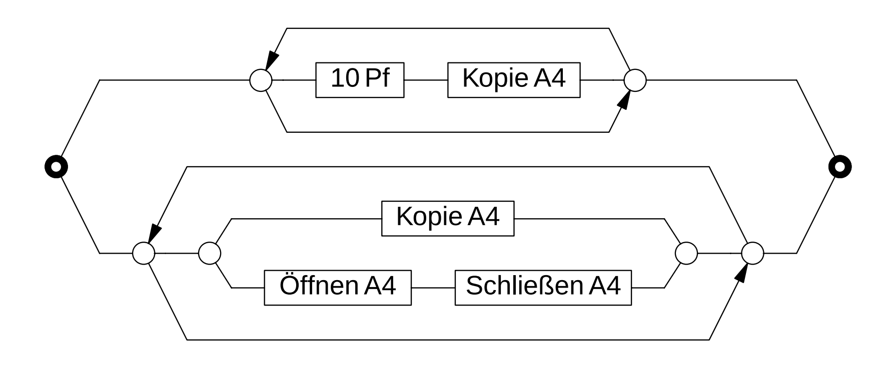


Abbildung 3.20: Beispiel einer Synchronisation

⟨*ZZ*, Kopie A4, *ZZ*, Öffnen A4, *ZZ*, Schließen A4, *ZZ*⟩,
⟨*ZZ*, Öffnen A4, *ZZ*, Schließen A4, *ZZ*, Kopie A4, *ZZ*⟩, ... }.

Bildet man den Durchschnitt dieser beiden Mengen, so erhält man die folgende Menge der vollständigen Worte des Ausdrucks $x \equiv y \circ z$:

$\Phi(y \circ z) = \Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^* = \{$ ⟨⟩, ⟨10 Pf, Kopie A4⟩, ⟨Öffnen A4, Schließen A4⟩,
⟨Öffnen A4, Schließen A4, 10 Pf, Kopie A4⟩, ⟨Öffnen A4, 10 Pf, Schließen A4, Kopie A4⟩,
⟨10 Pf, Öffnen A4, Schließen A4, Kopie A4⟩, ⟨10 Pf, Kopie A4, Öffnen A4, Schließen A4⟩, ... }.

Die Worte ⟨Öffnen A4, 10 Pf, Kopie A4, Schließen A4⟩ und ⟨10 Pf, Öffnen A4, Kopie A4, Schließen A4⟩ gehören aber beispielsweise nicht zur Menge $\Phi(y \circ z)$, da die Menge $\Phi(z) \otimes \kappa_x(z)^*$ keine Worte enthält, bei denen die Aktion Kopie A4 zwischen einer Aktion Öffnen A4 und der zugehörigen Aktion Schließen A4 steht.

Nach demselben Schema läßt sich auch die Menge der partiellen Worte des Ausdrucks $y \circ z$ bestimmen.

Anmerkung: Obwohl die formale Definition der Synchronisation, ebenso wie ihre Anwendung auf einen konkreten Ausdruck, auf den ersten Blick umständlich und „unnatürlich“ wirken mag, entspricht das Resultat genau dem intuitiv erwarteten Verhalten. Würde man den Synchronisationsoperator $\circ$ im obigen Beispiel durch eine Konjunktion $\bullet$ ersetzen, deren Definition rein formal wesentlich einfacher und „natürlicher“ wirkt, so würde der Graph nur noch Aktionen akzeptieren, die beiden Zweigen der Kopplung *gemeinsam* sind. Da die einzige derartige Aktion Kopie A4 aber erst im Anschluß an die Aktion 10 Pf durchlaufen werden kann, würde der Graph in Wirklichkeit überhaupt nichts mehr akzeptieren!

Hätte man den Operator $\circ$ nicht zur Verfügung (wie z. B. in *Synchronisierungsausdrücken*, vgl. § 6.2.2), so müßte man die oben beschriebene Transformation in eine Konjunktion als Graphautor selbst vornehmen, um einen Ausdruck zu erhalten, der *intuitiv* der logischen Und-Verknüpfung der beiden Zweige entspricht [Guo96]. Aus diesem Grund wird die Synchronisation gelegentlich auch als *intuitive Konjunktion* bezeichnet, deren praktischer Nutzen − insbesondere bei der *modularen Kombination* von Ausdrücken (vgl. § 2.3.2.4 und § 2.8.2.4) − weit über den der eigentlichen Konjunktion hinausgeht.

3.3.2 Abgeleitete Ausdrücke

3.3.2.1 Multiplikatorausdrücke

Definitionen

Ein *Multiplikatorausdruck* ist entweder ein *sequentieller Multiplikatorausdruck* $\overset{n}{\circlearrowleft} y$ oder ein *paralleler Multiplikatorausdruck* $\overset{n}{\circledcirc} y$ mit einem *Faktor* $n \in \mathbb{N}$ und einem beliebigen Ausdruck y, der als *Rumpf* des Multiplikatorausdrucks bezeichnet wird. Abbildung 3.21 zeigt die zugehörigen Interaktionsgraphen, die als Mehrfach-Ausführung (oben) bzw. Mehrfach-Verzweigung (unten) bezeichnet werden.

Multiplikatorausdrücke werden wie folgt auf elementare Interaktionsausdrücke zurückgeführt:

$$\overset{1}{\circlearrowleft} y = \overset{1}{\circledcirc} y = y,$$

$$\overset{n}{\circlearrowleft} y = \left(\overset{n-1}{\circlearrowleft} y\right) - y \quad \text{und} \quad \overset{n}{\circledcirc} y = \left(\overset{n-1}{\circledcirc} y\right) \circledcirc y \quad \text{für} \quad n > 1.$$

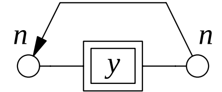


Abbildung 3.21: Multiplikatorausdrücke $\overset{n}{\circlearrowleft}\, y$ und $\overset{n}{\circledcirc}\, y$

Auf diese Weise ist ihre Semantik implizit festgelegt, ohne daß die Menge ihrer vollständigen und partiellen Worte explizit definiert werden muß.

Beispiel

Der Multiplikatorausdruck $\overset{2}{\circledcirc}\,(\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM}))$ kann wie folgt zu einem elementaren Ausdruck expandiert werden (vgl. Abb. 3.22):

$$
\begin{aligned}
& \overset{2}{\circledcirc}\,(\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})) \\
= & \left(\overset{1}{\circledcirc}\,(\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM}))\right) \circledcirc (\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})) \\
= & (\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})) \circledcirc (\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM})).
\end{aligned}
$$

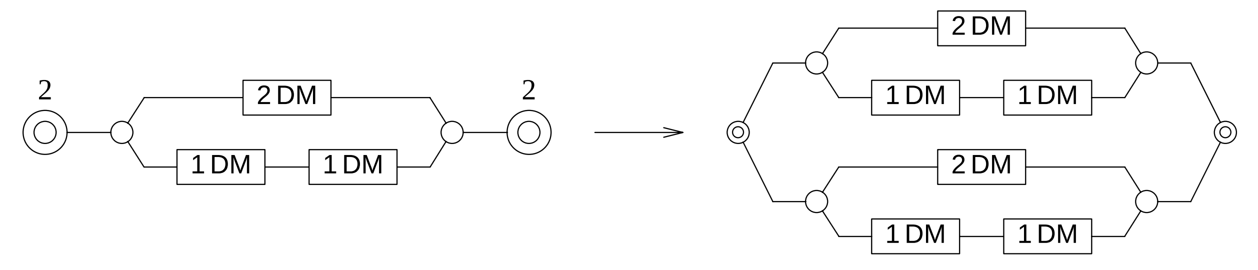


Abbildung 3.22: Expansion des Multiplikatorausdrucks $\overset{2}{\circledcirc}\,(\text{2 DM} \bigcirc (\text{1 DM} - \text{1 DM}))$

Anmerkungen

Rein formal sind Multiplikatoren lediglich „syntactic sugar“, da sie stets gemäß den obigen Definitionen durch elementare Operatoren ersetzt werden können. Daher werden sie im weiteren Verlauf dieses Kapitels auch nicht mehr berücksichtigt, d. h. sie werden quasi aus der Menge Ξ aller Interaktionsausdrücke entfernt. Für praktische Anwendungen sprechen allerdings einige wichtige Gründe für ihre Verwendung:

- Ein Multiplikator-Ausdruck ist häufig erheblich *kompakter, übersichtlicher* und *änderungsfreundlicher* als der äquivalente expandierte Ausdruck, insbesondere wenn der Faktor n groß ist.
- Das Wissen, daß bei einem Multiplikator-Ausdruck mehrmals *derselbe* Teilausdruck mit sich selbst verknüpft wird, kann u. U. gewinnbringend für eine effiziente Implementierung des Ausdrucks verwendet werden (vgl. § 4.5.5).

- Je nach Anwendung, kann es sich bei einem Faktor möglicherweise um eine *symbolische Konstante* oder eine *Variable* handeln, deren konkreter Wert entweder an anderer Stelle vereinbart oder aber erst zur Laufzeit berechnet wird (vgl. z. B. § 5.3.2.2, Fußnote). In diesem Fall ist eine expandierte Darstellung des Ausdrucks prinzipiell nicht möglich.

3.3.2.2 Makros

Makros (oder Abkürzungen und Schablonen) stellen ein weiteres nützliches Hilfsmittel zur Verbesserung der Lesbarkeit und Änderungsfreundlichkeit von Ausdrücken dar (vgl. § 2.2.4.4). Darüber hinaus unterstützen sie die *Wiederverwendbarkeit* von (Teil-)Ausdrücken sowie das wichtige Konzept der *Abstraktion*, indem sie eine Trennung von *abstraktem Konzept* (wie z. B. wechselseitiger Ausschluß) und *konkreter Implementierung* des Konzepts (z. B. als sequentielle Iteration über eine Disjunktion) erlauben (vgl. § 2.3.3.1). Diese Trennung ermöglicht es auch, daß Makros für bestimmte anwendungsspezifische Problemstellungen, wie z. B. die temporale Beziehung zwischen zwei Untersuchungsarten (vgl. § 2.7.4.4), von einem *Experten* erstellt und anschließend − ohne Kenntnis ihrer möglicherweise komplexen internen Struktur − von einem weniger geübten *Laien* verwendet werden können (vgl. auch § 2.8.2.2 und § 5.4.6).

Ungeachtet dieser, für den praktischen Einsatz von Interaktionsausdrücken äußerst wichtigen Faktoren, bieten Makros − rein formal betrachtet − keinerlei zusätzliche Funktionalität oder Ausdrucksmächtigkeit, da Makroaufrufe in einem Ausdruck (bzw. Abkürzungen oder Schablonen in einem Graphen) stets durch „intelligenten Textersatz" eliminiert werden können.[7] Aus diesem Grund wird an dieser Stelle nicht näher auf die konkrete syntaktische Notation von Makrodefinitionen und -aufrufen eingegangen. Außerdem werden Makros im folgenden − ebenso wie Multiplikatorausdrücke − nicht weiter berücksichtigt.

3.3.3 Quantorausdrücke

3.3.3.1 Definitionen

Ein *Quantorausdruck* ist ein Ausdruck $\underset{p}{\bigcirc}\, y$ (vgl. Abb. 3.23) mit einem Operator $\bigcirc \in \{ ○, ◎, ⭘, ● \}$, einem formalen Parameter $p \in \Pi$, der in diesem Zusammenhang auch als *Quantorparameter* bezeichnet wird, und einem beliebigen Teilausdruck y, der als *Rumpf* des Quantorausdrucks bzw. als *Wirkungsbereich* des Quantors $\underset{p}{\bigcirc}$ bezeichnet wird. Typischerweise hängt der Ausdruck y vom Quantorparameter p ab, d. h. er enthält in der Regel *abstrakte Aktionen* $a = [a_0, a_1, \ldots, a_n] \in \Gamma$, die p als Argument besitzen, d. h. für die $p \in \{ a_1, \ldots, a_n \}$ gilt.

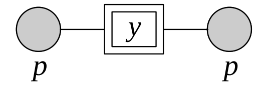


Abbildung 3.23: Quantorausdruck $\underset{p}{\bigcirc}\, y$

Der Quantorrumpf y kann nach Belieben weitere Quantorausdrücke enthalten; allerdings wird zur Vereinfachung angenommen, daß keiner dieser inneren Quantoren denselben Parameter p benutzt, d. h. daß Quantorparameter *eindeutig* sind. Da Parameter ggf. umbenannt werden können, stellt dies keine echte Einschränkung dar. Außerdem kann diese Vereinbarung in einer praktischen Implementie-

[7] Man beachte in diesem Zusammenhang, daß Makros nicht rekursiv sein dürfen (vgl. § 2.2.4.4 sowie § 2.8.2.5).

rung durch geeignete *Sichtbarkeitsregeln* (engl. scope rules) ersetzt werden, wie man sie von blockorientierten Programmiersprachen kennt (vgl. § 4.2.4).

Konkretisierte Aktionen und Ausdrücke

Ersetzt man in einer abstrakten Aktion $a = [a_0, a_1, \ldots, a_n] \in \Gamma$ jedes Vorkommen eines formalen Parameters $p \in \Pi$ durch einen konkreten Wert $\omega \in \Omega$, so erhält man die *konkretisierte Aktion* a_p^ω, die formal wie folgt definiert ist:

$$a_p^\omega = [a_0, a_1', \ldots, a_n'] \quad \text{mit} \quad a_i' = \begin{cases} \omega, & \text{falls } a_i = p, \\ a_i & \text{sonst,} \end{cases} \qquad \text{für } i = 1, \ldots, n.$$

In ähnlicher Weise erhält man einen *konkretisierten Ausdruck* x_p^ω, wenn man jede Aktion a von x durch ihre konkretisierte Aktion a_p^ω ersetzt, das heißt:

$$x_p^\omega \equiv \begin{cases} a_p^\omega & \text{für } x \equiv a & \text{mit } a \in \Gamma, \\ \bigcirc\, y_p^\omega & \text{für } x \equiv \bigcirc\, y & \text{mit } \bigcirc \in \{ \smile\!\!\!\circlearrowright, \ominus, \circledcirc \}, \\ y_p^\omega \bigcirc z_p^\omega & \text{für } x \equiv y \bigcirc z & \text{mit } \bigcirc \in \{ -, \circ, \circledcirc, \boldsymbol{\circ}, \bullet \}, \\ \underset{q}{\bigcirc}\, y_p^\omega & \text{für } x \equiv \underset{q}{\bigcirc}\, y & \text{mit } \bigcirc \in \{ \circ, \circledcirc, \boldsymbol{\circ}, \bullet \}. \end{cases}$$

Anschauliche Interpretation von Quantorausdrücken

Anschaulich entspricht ein Quantorausdruck $x \equiv \underset{p}{\bigcirc}\, y$ einem unendlichen Ausdruck

$$\underset{\omega \in \Omega}{\bigcirc} y_p^\omega \equiv \overset{\infty}{\underset{i=1}{\bigcirc}} y_p^{\omega_i} \equiv y_p^{\omega_1} \bigcirc y_p^{\omega_2} \bigcirc y_p^{\omega_3} \bigcirc \ldots \qquad \text{(vgl. Abb. 3.24)}$$

mit $\{ \omega_1, \omega_2, \ldots \} = \Omega$.[8] Ausgehend von dieser Vorstellung, werden die Mengen $\Phi(x)$, $\Psi(x)$ und $\alpha(x)$ durch geeignete Verallgemeinerungen der entsprechenden Definitionen für elementare Ausdrücke gemäß Tab. 3.25 definiert.

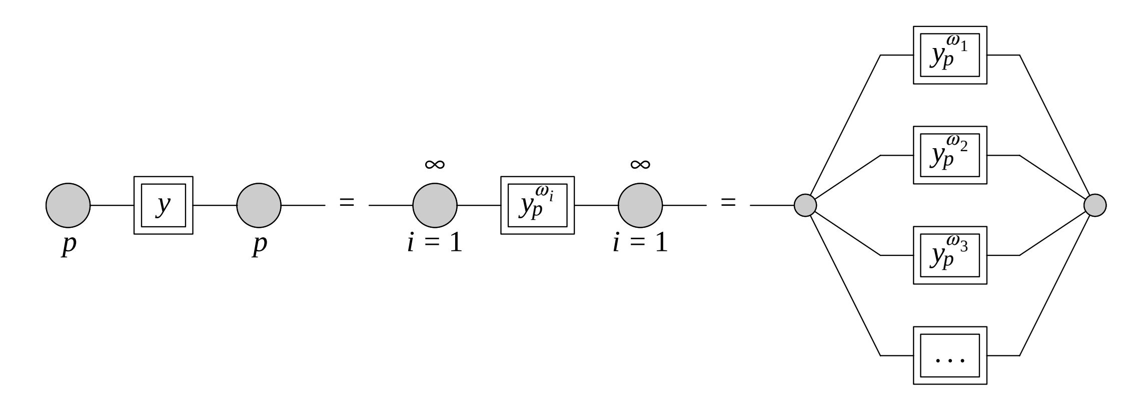


Abbildung 3.24: Anschauliche Interpretation eines Quantorausdrucks als unendlicher Ausdruck

[8] Allerdings könnte die Menge Ω auch *überabzählbar* sein, so daß eine Aufzählung ihrer Elemente $\omega_1, \omega_2, \ldots$ nicht mehr möglich ist.

x	$\Phi(x)$	$\Psi(x)$	$\alpha(x)$
$\underset{p}{\bigcirc}\, y$	$\bigcup_{\omega \in \Omega} \Phi\left(y_p^\omega\right)$	$\bigcup_{\omega \in \Omega} \Psi\left(y_p^\omega\right)$	$\bigcup_{\omega \in \Omega} \alpha\left(y_p^\omega\right)$
$\underset{p}{\circledcirc}\, y$	$\bigotimes_{\omega \in \Omega} \Phi\left(y_p^\omega\right)$	$\bigotimes_{\omega \in \Omega} \Psi\left(y_p^\omega\right)$	$\bigcup_{\omega \in \Omega} \alpha\left(y_p^\omega\right)$
$\underset{p}{\boldsymbol{\circ}}\, y$	$\bigcap_{\omega \in \Omega} \Phi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$	$\bigcap_{\omega \in \Omega} \Psi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$	$\bigcup_{\omega \in \Omega} \alpha\left(y_p^\omega\right)$
$\underset{p}{\bullet}\, y$	$\bigcap_{\omega \in \Omega} \Phi\left(y_p^\omega\right)$	$\bigcap_{\omega \in \Omega} \Psi\left(y_p^\omega\right)$	$\bigcup_{\omega \in \Omega} \alpha\left(y_p^\omega\right)$

Tabelle 3.25: Definition der Mengen $\Phi(x)$, $\Psi(x)$ und $\alpha(x)$ für Quantorausdrücke

3.3.3.2 Erläuterungen

Die *Vereinigung* bzw. der *Durchschnitt* unendlich vieler Mengen ist wie üblich als

$$\bigcup_{\omega \in \Omega} U_\omega = \{\, w \mid \exists\, \omega \in \Omega \colon w \in U_\omega \,\}$$

bzw.

$$\bigcap_{\omega \in \Omega} U_\omega = \{\, w \mid \forall\, \omega \in \Omega \colon w \in U_\omega \,\}$$

definiert, während die *Verschränkung* $\bigotimes_{\omega \in \Omega} U_\omega$ unendlich vieler Mengen in § 3.2.2.4 eingeführt wurde. Gemäß dieser Definition sind die Mengen $\Phi(x)$ und $\Psi(x)$ für $x \equiv \underset{p}{\circledcirc}\, y$ genau dann verschieden von der leeren Menge, wenn die Mengen $\Phi\left(y_p^\omega\right)$ bzw. $\Psi\left(y_p^\omega\right)$ für alle Werte $\omega \in \Omega$ das leere Wort enthalten. Für die Mengen $\Psi\left(y_p^\omega\right)$ ist dies gemäß § 3.4.4 grundsätzlich der Fall, d. h. es gilt stets $\Psi(x) \neq \emptyset$. Für die Mengen $\Phi\left(y_p^\omega\right)$ wird in § 3.4.6 ein syntaktisches Kriterium für den Ausdruck y genannt, das darüber entscheidet, ob das leere Wort in allen diesen Mengen enthalten ist oder nicht.

Die Mengen $\kappa_x\left(y_p^\omega\right)$ werden analog zu § 3.3.1.10 als

$$\kappa_x\left(y_p^\omega\right) = \alpha(x) \setminus \alpha\left(y_p^\omega\right) \qquad \text{für } \omega \in \Omega$$

definiert.

3.3.3.3 Anmerkungen

Für praktische Anwendungen sind vor allem *Disjunktions-Quantorausdrücke* (oder Für-ein-Verzweigungen) $\underset{p}{\bigcirc}\, y$ und *parallele Quantorausdrücke* (oder Für-alle-Verzweigungen) $\underset{p}{\circledcirc}\, y$ relevant, während Synchronisations- und Konjunktions-Quantorausdrücke primär aus Gründen der konzeptionellen Vollständigkeit eingeführt wurden. Somit existiert zu jedem *kommutativen* binären Operator $\bigcirc \in \{\, \circ, \circledcirc, \boldsymbol{\circ}, \bullet \,\}$ (vgl. § 3.4.7) ein zugehöriger Quantor $\underset{p}{\bigcirc}$.

Die sequentielle Komposition nimmt unter den binären Operatoren insofern eine Sonderstellung ein, als sie offensichtlich *nicht* kommutativ ist. Aus diesem Grund kann sie nicht ohne weiteres zu einem Quantor verallgemeinert werden, da sowohl in der anschaulichen Interpretation gemäß Abb. 3.24 als auch in den formalen Definitionen in Tab. 3.25 implizit die Kommutativität des jeweiligen Operators $\bigcirc$ (bzw. der Mengenoperationen Vereinigung, Durchschnitt und Verschränkung; vgl. § 3.4.7) ausgenutzt wird, die gewährleistet, daß die Bedeutung des Quantorausdrucks $\underset{p}{\bigcirc}\, y$ unabhängig von einer bestimmten *Reihenfolge* oder Anordnung der Werte $\omega \in \Omega$ ist.

3.3.3.4 Beispiele

Bezeichne im folgenden abrufen(p, u) die Tätigkeit Patient p für Untersuchung u abrufen und untersuchen(p, u) die Tätigkeit Untersuchung u für Patient p durchführen. Im Gegensatz zu § 2.7 wird zur Vereinfachung angenommen, daß es sich bei diesen Tätigkeiten um *punktuelle Aktionen* und nicht um zeitlich ausgedehnte Aktivitäten handelt. Dann besitzt der Ausdruck

$$\circlearrowright \bigcirc_{u} (\text{abrufen(Meier}, u) - \text{untersuchen(Meier}, u)) \qquad \text{(vgl. Abb. 3.26)}$$

die folgenden vollständigen Worte:

$$\begin{aligned}
& \Phi\Big(\circlearrowright \bigcirc_{u} (\text{abrufen(Meier}, u) - \text{untersuchen(Meier}, u))\Big) \\
= {} & \Phi\Big(\bigcirc_{u} (\text{abrufen(Meier}, u) - \text{untersuchen(Meier}, u))\Big)^* \\
= {} & \Big(\bigcup_{\omega \in \Omega} \Phi(\text{abrufen(Meier}, \omega) - \text{untersuchen(Meier}, \omega))\Big)^* \\
= {} & \Big(\bigcup_{\omega \in \Omega} \{ \langle \text{abrufen(Meier}, \omega), \text{untersuchen(Meier}, \omega) \rangle \}\Big)^* \\
= {} & \{\langle \text{abrufen(Meier, sono), untersuchen(Meier, sono)} \rangle, \\
& \ \langle \text{abrufen(Meier, endo), untersuchen(Meier, endo)} \rangle, \ldots\}^*.
\end{aligned}$$

Diese Menge enthält z. B. die Worte:

⟨⟩,

⟨abrufen(Meier, sono), untersuchen(Meier, sono)⟩,
⟨abrufen(Meier, endo), untersuchen(Meier, endo)⟩,

⟨abrufen(Meier, sono), untersuchen(Meier, sono),
abrufen(Meier, endo), untersuchen(Meier, endo)⟩,
⟨abrufen(Meier, endo), untersuchen(Meier, endo),
abrufen(Meier, radio), untersuchen(Meier, radio)⟩,

⟨abrufen(Meier, radio), untersuchen(Meier, radio),
abrufen(Meier, endo), untersuchen(Meier, endo),
abrufen(Meier, sono), untersuchen(Meier, sono)⟩,

usw.

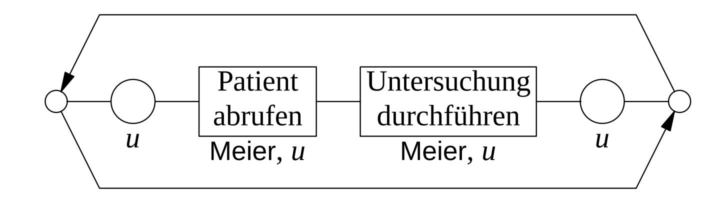


Abbildung 3.26: Beispiel eines Disjunktions-Quantorausdrucks

Ersetzt man im obigen Ausdruck den konkreten Wert Meier $\in \Omega$ durch den formalen Parameter $p \in \Pi$ und quantifiziert anschließend mit Hilfe eines parallelen Quantors über alle Patienten p, so erhält man den Ausdruck

$$\underset{p}{\circledcirc} \circlearrowright \underset{u}{\bigcirc} (\text{abrufen}(p, u) - \text{untersuchen}(p, u)) \qquad \text{(vgl. Abb. 3.27).}$$

Da die Menge $\Phi\left(\circlearrowright \underset{u}{\bigcirc} (\text{abrufen}(\omega, u) - \text{untersuchen}(\omega, u))\right)$ für jeden Wert $\omega \in \Omega$ das leere Wort enthält, besitzt dieser Ausdruck die folgenden vollständigen Worte:

$$\Phi\left(\underset{p}{\circledcirc} \circlearrowright \underset{u}{\bigcirc} (\text{abrufen}(p, u) - \text{untersuchen}(p, u))\right)$$

$$= \bigotimes_{\omega \in \Omega} \Phi\left(\circlearrowright \underset{u}{\bigcirc} (\text{abrufen}(\omega, u) - \text{untersuchen}(\omega, u))\right)$$

$$= \bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} \Phi\left(\circlearrowright \underset{u}{\bigcirc} \big(\text{abrufen}(\omega_i, u) - \text{untersuchen}(\omega_i, u)\big)\right)$$

$$= \bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} \Phi\left(\underset{u}{\bigcirc} \big(\text{abrufen}(\omega_i, u) - \text{untersuchen}(\omega_i, u)\big)\right)^*$$

$$= \bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} \left(\bigcup_{\pi \in \Omega} \Phi\Big(\big(\text{abrufen}(\omega_i, \pi) - \text{untersuchen}(\omega_i, \pi)\big)\Big)\right)^*.$$

$$= \bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} \left(\bigcup_{\pi \in \Omega} \left\{ \langle \text{abrufen}(\omega_i, \pi), \text{untersuchen}(\omega_i, \pi) \rangle \right\}\right)^*.$$

Diese Menge enthält z. B. die Worte:

⟨⟩,

⟨abrufen(Meier, sono), untersuchen(Meier, sono)⟩,
⟨abrufen(Mayr, endo), untersuchen(Mayr, endo)⟩,

⟨abrufen(Maier, sono), abrufen(Mayr, endo),
untersuchen(Mayr, endo), untersuchen(Maier, sono)⟩,
⟨abrufen(Meyer, radio), abrufen(Meier, sono),
untersuchen(Meyer, radio), untersuchen(Meier, sono)⟩,

⟨abrufen(Mair, sono), abrufen(Maier, endo), untersuchen(Maier, endo),
abrufen(Mayer, radio), untersuchen(Mair, sono), untersuchen(Mayer, radio)⟩,

usw.

Durch die Verschachtelung von parallelem Quantor $\underset{p}{\circledcirc}$, Iterationsoperator $\circlearrowright$ und Disjunktions-Quantor $\underset{u}{\bigcirc}$ sind die Kombinationsmöglichkeiten hier jedoch so vielfältig, daß es kaum möglich ist, die Menge der zulässigen Worte durch einige willkürlich herausgegriffene Beispiele ausreichend zu charakterisieren. Für ein intuitives Verständnis des Ausdrucks ist dies allerdings auch nicht erforderlich.

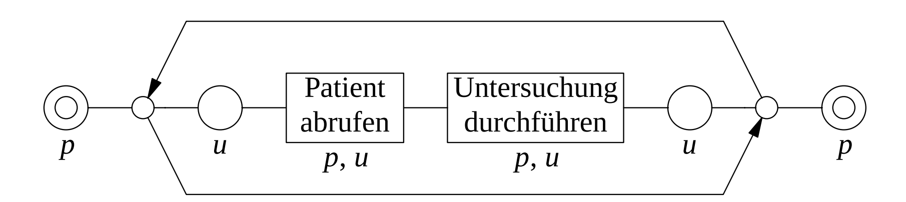


Abbildung 3.27: Beispiel eines parallelen Quantorausdrucks

Anmerkung: Die Beispiele zeigen, wie durch die Anwendung der Quantordefinitionen gemäß Tab. 3.25 die Parameter p und u der Aktionen abrufen und untersuchen sukzessive eliminiert werden. Am Ende verbleiben nur konkrete Aktionen $\in \Sigma$, für die die Menge der partiellen und vollständigen Worte auf natürliche Art und Weise definiert ist (vgl. § 3.3.1.1). Die eher ungewöhnlichen Definitionen $\Phi(a) = \varnothing$ und $\Psi(a) = \{ \langle\rangle \}$ für $a \in \Gamma \setminus \Sigma$ kommen nur dann zur Anwendung, wenn ein Parameter *global ungebunden*, d. h. nicht durch einen umgebenden Quantor gebunden ist. Derartige Ausdrücke kommen in praktischen Anwendungen jedoch nicht vor.[9]

3.3.4 Zusammenfassung

Da die Definitionen der drei charakteristischen Mengen $\Phi(x)$, $\Psi(x)$ und $\alpha(x)$ eines Interaktionsausdrucks x im nachfolgenden Abschnitt 3.4 häufig benötigt werden, sind sie in Tab. 3.28 nochmals für sämtliche Kategorien von Ausdrücken zusammengefaßt.

x	$\Phi(x)$	$\Psi(x)$	$\alpha(x)$
a	$\{ \langle a\rangle \} \cap \Sigma^*$	$\{ \langle\rangle, \langle a\rangle \} \cap \Sigma^*$	$\{ a \}$
$\text{◡} y$	$\Phi(y) \cup \{ \langle\rangle \}$	$\Psi(y)$	$\alpha(y)$
$y - z$	$\Phi(y)\,\Phi(z)$	$\Psi(y) \cup \Phi(y)\,\Psi(z)$	$\alpha(y) \cup \alpha(z)$
$\ominus y$	$\Phi(y)^*$	$\Phi(y)^*\,\Psi(y)$	$\alpha(y)$
$y \circledcirc z$	$\Phi(y) \otimes \Phi(z)$	$\Psi(y) \otimes \Psi(z)$	$\alpha(y) \cup \alpha(z)$
$\text{⦾} y$	$\Phi(y)\#$	$\Psi(y)\#$	$\alpha(y)$
$y \bigcirc z$	$\Phi(y) \cup \Phi(z)$	$\Psi(y) \cup \Psi(z)$	$\alpha(y) \cup \alpha(z)$
$y \bullet z$	$\Phi(y) \cap \Phi(z)$	$\Psi(y) \cap \Psi(z)$	$\alpha(y) \cup \alpha(z)$
$y \boldsymbol{\circ} z$	$\Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^*$	$\Psi(y) \otimes \kappa_x(y)^* \cap \Psi(z) \otimes \kappa_x(z)^*$	$\alpha(y) \cup \alpha(z)$
$\underset{p}{\bigcirc} y$	$\bigcup_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$	$\bigcup_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	$\bigcup_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$
$\underset{p}{\circledcirc} y$	$\bigotimes_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$	$\bigotimes_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	$\bigcup_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$
$\underset{p}{\boldsymbol{\circ}} y$	$\bigcap_{\omega\in\Omega} \Phi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$	$\bigcap_{\omega\in\Omega} \Psi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$	$\bigcup_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$
$\underset{p}{\bullet} y$	$\bigcap_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$	$\bigcap_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	$\bigcup_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$

Tabelle 3.28: Definition der Mengen $\Phi(x)$, $\Psi(x)$ und $\alpha(x)$ für beliebige Interaktionsausdrücke

3.4 Eigenschaften von Interaktionsausdrücken

Nachdem Interaktionsausdrücke nun präzise und vollständig definiert sind, sollen im folgenden verschiedene *formale Eigenschaften* bzw. *Umformungsregeln* für sie bewiesen werden. Viele dieser Regeln sind intuitiv einleuchtend und wurden z. T. bereits ohne Beweis in Kapitel 2 erwähnt und verwendet. So wird beispielsweise in § 2.2.2.2 implizit die Assoziativität der Disjunktion vorausgesetzt, während in § 2.2.4.2 explizit die Assoziativität und Kommutativität der parallelen Komposition ausgenutzt wird.

Die Beweise sind größtenteils einfacher Natur und werden daher nicht immer vollständig ausgeführt. Lediglich im Kontext von Quantoren (§ 3.4.9) sind sie zum Teil etwas umfangreicher und schwieriger.

[9] Wie in der Fußnote in § 3.3.1.1 erwähnt wurde, ist es jedoch einfacher, sie *nicht* grundsätzlich zu verbieten.

Die Anordnung der nachfolgenden Abschnitte ist zu einem großen Teil durch explizite oder implizite Abhängigkeiten zwischen den einzelnen Aussagen und Beweisen bestimmt. Beispielsweise werden in § 3.4.11 Resultate der Abschnitte 3.4.6 und 3.4.8 verwendet; Abschnitt 3.4.8 wiederum stützt sich auf Aussagen aus § 3.4.5; dieser schließlich hängt von § 3.4.2 ab. Sofern die resultierende „topologische Sortierung" der Regeln noch Freiheitsgrade offenließ, wurde versucht, logisch zusammengehörende Aussagen möglichst aufeinanderfolgend vorzustellen. So findet man beispielsweise die grundlegenden Aussagen über Assoziativität, Kommutativität und Idempotenz von Operatoren nacheinander in den Abschnitten 3.4.7, 3.4.8 und 3.4.9, die aufgrund von Abhängigkeitsbeziehungen aber erst relativ spät formuliert werden können.

Da abgeleitete Ausdrücke (Multiplikatoren und Makros) stets gemäß ihrer Definition eliminiert werden können (vgl. § 3.3.2), werden im folgenden nur elementare und Quantorausdrücke betrachtet.

3.4.1 Äquivalenz von Ausdrücken

3.4.1.1 Definitionen

1. Zwei Ausdrücke x_1 und x_2 heißen *klassisch gleich* oder *äquivalent* (in Zeichen $x_1 \sim x_2$), wenn sie dieselben *vollständigen* Worte akzeptieren, d. h. wenn gilt:

$$\Phi(x_1) = \Phi(x_2).$$

2. Zwei Ausdrücke x_1 und x_2 heißen *isoliert gleich* oder *äquivalent* (in Zeichen $x_1 \approx x_2$), wenn sie dieselben *vollständigen* und *partiellen* Worte akzeptieren und dasselbe *Alphabet* besitzen, d. h. wenn gilt:

$$\Phi(x_1) = \Phi(x_2), \quad \Psi(x_1) = \Psi(x_2) \quad \text{und} \quad \alpha(x_1) = \alpha(x_2).$$

3. Zwei Ausdrücke x_1 und x_2 heißen *(umfassend) gleich* oder *äquivalent* (in Zeichen $x_1 = x_2$), wenn für beliebige Parameter $p_1 \neq \ldots \neq p_n \in \Pi$ und zugehörige Werte $\omega_1, \ldots, \omega_n \in \Omega$ gilt:

$$(x_1)^{\omega_1, \ldots, \omega_n}_{p_1, \ldots, p_n} = (x_2)^{\omega_1, \ldots, \omega_n}_{p_1, \ldots, p_n} \qquad \text{mit} \qquad x^{\omega_1, \ldots, \omega_n}_{p_1, \ldots, p_n} \equiv \begin{cases} x & \text{für } n = 0, \\ \left(x^{\omega_1, \ldots, \omega_{n-1}}_{p_1, \ldots, p_{n-1}}\right)^{\omega_n}_{p_n} & \text{für } n > 0, \end{cases}$$

d. h. wenn sämtliche *Konkretisierungen* von x_1 und x_2 isoliert äquivalent sind. (Für $n = 0$ ergibt sich insbesondere, daß x_1 und x_2 selbst isoliert äquivalent sein müssen.)

3.4.1.2 Anmerkungen

1. Die klassische Äquivalenz von Ausdrücken ist lediglich für den Vergleich von Interaktionsausdrücken mit „klassischen" Formalismen wie z. B. regulären Ausdrücken oder kontextfreien Grammatiken interessant, für die üblicherweise nur vollständige Worte betrachtet werden (vgl. § 3.5).
Da in diesem Zusammenhang nur konkrete Ausdrücke ohne Parameter betrachtet werden, ist eine Unterscheidung von isolierter und umfassender Äquivalenz hier nicht erforderlich.

2. Sind zwei Ausdrücke x_1 und x_2 umfassend äquivalent, so kann x_1 in jedem Kontext durch x_2 ersetzt werden und umgekehrt.
Wie in § 2.5.2.1 (Fußnote) gezeigt wurde, müssen x_1 und x_2 hierfür nicht nur dieselben partiellen und vollständigen Worte, sondern auch dasselbe Alphabet besitzen. Ist letzteres nicht der Fall, sind die Ausdrücke nur *fast äquivalent*.

3. Gilt eine isolierte Gleichheitsaussage

$$f(x_1, \ldots, x_n) \approx g(x_1, \ldots, x_n)$$

für *alle* Ausdrücke $x_1, \ldots, x_n \in \Xi$, so gilt offensichtlich auch

$$f(x'_1, \ldots, x'_n) \approx g(x'_1, \ldots, x'_n)$$

für beliebige Konkretisierungen $x_i' \equiv x_{p_1, \ldots, p_n}^{\omega_1, \ldots, \omega_n}$ der Ausdrücke x_i ($i = 1, \ldots, n$). Daraus folgt aber gemäß Definition 3 unmittelbar die Gültigkeit der umfassenden Gleichheitsaussage

$$f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$$

für alle Ausdrücke $x_1, \ldots, x_n \in \Xi$.
Daher genügt es, zum Beweis der umfassenden Gleichheitsaussage die Richtigkeit der isolierten Gleichheitsaussage für alle Ausdrücke $x_1, \ldots, x_n \in \Xi$ zu zeigen.

3.4.1.3 Beispiel

Die Ausdrücke

$$x_1 \equiv a(p) \bullet a(q) \quad \text{und} \quad x_2 \equiv a(p) \circ a(q) \quad \text{mit} \quad p \neq q \in \Pi$$

sind isoliert äquivalent, denn es gilt wegen $a(p), a(q) \notin \Sigma$:

$$\Phi(x_1) = \emptyset = \Phi(x_2), \quad \Psi(x_1) = \{ \langle\rangle \} = \Psi(x_2) \quad \text{und} \quad \alpha(x_1) = \{ a(p), a(q) \} = \alpha(x_2).$$

Für $\omega \neq \pi \in \Omega$ gilt jedoch:

$$x_1' \equiv (x_1)_{p,\,q}^{\omega,\,\pi} = a(\omega) \bullet a(\pi) \quad \text{mit}$$
$$\Phi(x_1') = \Phi(a(\omega)) \cap \Phi(a(\pi)) = \{ \langle a(\omega)\rangle \} \cap \{ \langle a(\pi)\rangle \} = \emptyset$$

und

$$x_2' \equiv (x_2)_{p,\,q}^{\omega,\,\pi} = a(\omega) \circ a(\pi) \quad \text{mit}$$
$$\Phi(x_2') = \Phi(a(\omega)) \cup \Phi(a(\pi)) = \{ \langle a(\omega)\rangle \} \cup \{ \langle a(\pi)\rangle \} = \{ \langle a(\omega)\rangle, \langle a(\pi)\rangle \} \neq \Phi(x_1'),$$

d. h. x_1 und x_2 sind nicht umfassend äquivalent. Dementsprechend kann x_1 normalerweise nicht durch x_2 ersetzt werden, ohne die Bedeutung eines übergeordneten Ausdrucks zu verändern.

3.4.2 Monotonie von Operatoren

3.4.2.1 Lemma

Die Mengenoperationen Vereinigung und Durchschnitt, Konkatenation und Verschränkung sowie sequentielle und parallele Hülle sind *monoton* und *abgeschlossen*, d. h. aus $U_1 \subseteq U_2 \subseteq A^*$ und $V_1 \subseteq V_2 \subseteq A^*$ (für eine Teilmenge $A \subseteq \Sigma$) folgt:

$$U_1 \cup V_1 \subseteq U_2 \cup V_2 \subseteq A^*,$$

$$U_1 \cap V_1 \subseteq U_2 \cap V_2 \subseteq A^*,$$

$$U_1 V_1 \subseteq U_2 V_2 \subseteq A^*,$$

$$U_1 \otimes V_1 \subseteq U_2 \otimes V_2 \subseteq A^*,$$

$$U_1{}^* \subseteq U_2{}^* \subseteq A^*,$$

$$U_1\# \subseteq U_2\# \subseteq A^*.$$

Beweis

Die Aussagen gehören entweder zum mathematischen Allgemeinwissen oder folgen unmittelbar aus den Definitionen der jeweiligen Operationen (vgl. § 3.2.2.2 und § 3.2.2.3).

3.4.2.2 Definitionen

1. Ein Ausdruck x_1 *akzeptiert höchstens die Worte*, die ein Ausdruck x_2 akzeptiert (in Zeichen $x_1 \subseteq x_2$), wenn für alle Konkretisierungen $x_i' \equiv (x_i)^{\omega_1, \ldots, \omega_n}_{p_1, \ldots, p_n}$ $(i = 1, 2)$ gilt:

$$\Phi(x_1') \subseteq \Phi(x_2') \quad \text{und} \quad \Psi(x_1') \subseteq \Psi(x_2').$$

2. Ein Ausdruck x_1 heißt (gleich oder) *restriktiver* als ein Ausdruck x_2 (in Zeichen $x_1 \leq x_2$), wenn er dasselbe Alphabet wie x_2 besitzt und höchstens die Worte akzeptiert, die x_2 akzeptiert, d. h. wenn für alle Konkretisierungen $x_i' \equiv (x_i)^{\omega_1, \ldots, \omega_n}_{p_1, \ldots, p_n}$ $(i = 1, 2)$ gilt:

$$\Phi(x_1') \subseteq \Phi(x_2'), \quad \Psi(x_1') \subseteq \Psi(x_2') \quad \text{und} \quad \alpha(x_1') = \alpha(x_2').$$

3. Gilt $x_1 \leq x_2$ und $x_1 \neq x_2$, so heißt x_1 *echt restriktiver* als x_2, in Zeichen $x_1 < x_2$.

3.4.2.3 Satz

1. Sämtliche Operatoren von Interaktionsausdrücken sind *monoton*, d. h. aus $y_1 \leq y_2$ und $z_1 \leq z_2$ folgt:

$$\bigcirc\, y_1 \leq \bigcirc\, y_2 \qquad \text{für } \bigcirc \in \{\, \curvearrowright, \ominus, \circledcirc \,\},$$
$$y_1 \bigcirc z_1 \leq y_2 \bigcirc z_2 \qquad \text{für } \bigcirc \in \{\, -, \circ, \circledcirc, \boldsymbol{\circ}, \bullet \,\},$$
$$\underset{p}{\bigcirc}\, y_1 \leq \underset{p}{\bigcirc}\, y_2 \qquad \text{für } \bigcirc \in \{\, \circ, \circledcirc, \boldsymbol{\circ}, \bullet \,\}.$$

2. Aus $x_1 \leq x_2$ und $x_2 \leq x_1$ folgt $x_1 = x_2$.

Beweis

1. Ebenso wie bei der Äquivalenz von Ausdrücken (§ 3.4.1.2) genügt es jeweils, die entsprechenden „isolierten Aussagen“ zu zeigen, da sich diese unmittelbar zu „umfassenden Aussagen“ verallgemeinern lassen.
   Die Behauptungen der isolierten Aussagen folgen unmittelbar aus den Definitionen der Operatoren (vgl. Tab. 3.28, § 3.3.4) zusammen mit der Monotonie der dort verwendeten Mengenoperationen (vgl. Lemma). Im Kontext der Synchronisation ist außerdem zu beachten, daß die Alphabete korrespondierender Teilausdrücke gleich sind (z. B. $\alpha(y_1) = \alpha(y_2)$) und somit auch korrespondierende Mengen $\kappa_x(\ldots)$ gleich sind.

2. Unmittelbar aus der Definition der Relationen $\leq$ und $=$.

3.4.2.4 Anmerkung

Würde man die Bedingung $\alpha(x_1') = \alpha(x_2')$ in der Definition von $x_1 \leq x_2$ zu $\alpha(x_1') \subseteq \alpha(x_2')$ abschwächen, so wäre die erste Behauptung des Satzes falsch, wie das folgende Beispiel zeigt.

Sei $y_1 \equiv a$, $y_2 \equiv a - \curvearrowright b$ und $z_1 \equiv z_2 \equiv b$. Dann gilt offensichtlich $\Phi(y_1) \subseteq \Phi(y_2)$, $\Psi(y_1) \subseteq \Psi(y_2)$ und $\alpha(y_1) \subseteq \alpha(y_2)$, aber nicht $\alpha(y_1) = \alpha(y_2)$. Außerdem gilt natürlich $z_1 \leq z_2$. Für die Ausdrücke $x_1 \equiv y_1 \boldsymbol{\circ} z_1$ und $x_2 \equiv y_2 \boldsymbol{\circ} z_2$ gilt nun:

$$\Phi(x_1) = \Phi(a \boldsymbol{\circ} b) = \{\, \langle a, b \rangle, \langle b, a \rangle \,\},$$

$$\Psi(x_1) = \Psi(a \boldsymbol{\circ} b) = \{\, \langle\rangle, \langle a \rangle, \langle b \rangle, \langle a, b \rangle, \langle b, a \rangle \,\} \quad \text{und}$$

$$\alpha(x_1) = \{\, a, b \,\}$$

bzw.

$$\Phi\big(x_2\big) = \Phi((a -_{\curvearrowright} b) \boldsymbol{\circ}\ b) = \{ \langle a, b \rangle \},$$
$$\Psi\big(x_2\big) = \Psi((a -_{\curvearrowright} b) \boldsymbol{\circ}\ b) = \{ \langle\rangle, \langle a \rangle, \langle a, b \rangle \} \qquad \text{und}$$
$$\alpha\big(x_2\big) = \{ a, b \},$$

d. h. x_2 ist echt restriktiver als x_1, obwohl y_1 abgesehen vom Alphabet echt restriktiver als y_2 ist.

3.4.3 Induktionsprinzip für Ausdrücke

3.4.3.1 Satz

Um die Richtigkeit einer Aussage $A(x)$ (wie z. B. $\Phi(x) \subseteq \Psi(x)$, vgl. § 3.4.4) für alle Ausdrücke $x \in \Xi$ zu beweisen, genügt es, ihre Richtigkeit für alle atomaren Ausdrücke $x \equiv a$ zu beweisen sowie für zusammengesetzte Ausdrücke $x \equiv \bigcirc\, y$, $x \equiv y \bigcirc z$ und $x \equiv \underset{p}{\bigcirc}\, y$ zu beweisen, daß die Richtigkeit von $A(x)$ aus der Richtigkeit von A für alle (ggf. konkretisierten) Teilausdrücke von x folgt, d. h.:

$$A(y) \;\Rightarrow\; A(\bigcirc\, y),$$
$$A(y) \wedge A(z) \;\Rightarrow\; A(y \bigcirc z),$$
$$A(y) \wedge \Big(\forall\, \omega \in \Omega{:}\ A\big(y_p^{\omega}\big)\Big) \;\Rightarrow\; A\Big(\underset{p}{\bigcirc}\, y\Big).$$

Beweis

1. Betrachte die Aussage $B(n)$:

   $A(x)$ ist für alle Ausdrücke x, die höchstens n Operatoren enthalten, richtig.

   Offensichtlich folgt die Richtigkeit von $A(x)$ für alle möglichen Ausdrücke x aus der Richtigkeit von $B(n)$ für alle $n \in \mathbb{N}_0$. Nach dem Prinzip der vollständigen Induktion für natürliche Zahlen folgt dies wiederum aus der Richtigkeit von $B(0)$ und der Richtigkeit der Implikation $B(n-1) \;\Rightarrow\; B(n)$ für alle $n \in \mathbb{N}$.

2. Da die atomaren Ausdrücke $x \equiv a$ genau die Ausdrücke mit null Operatoren darstellen, folgt die Richtigkeit der Aussage $B(0)$ aus der Richtigkeit der Aussage $A(x)$ für alle atomaren Ausdrücke $x \equiv a$.

3. Wenn ein zusammengesetzter Ausdruck $x \equiv \bigcirc\, y$, $x \equiv y \bigcirc z$ oder $x \equiv \underset{p}{\bigcirc}\, y$ genau n Operatoren enthält, dann enthalten seine Teilausdrücke y und ggf. z höchstens $n-1$ Operatoren. Daher ist die Implikation $B(n-1) \;\Rightarrow\; B(n)$ für alle $n \in \mathbb{N}$ richtig, wenn die Richtigkeit von $A(x)$ für alle zusammengesetzten Ausdrücke x aus der Richtigkeit von A für alle Teilausdrücke von x folgt.

3.4.3.2 Anwendung

Um im folgenden Aussagen für beliebige Ausdrücke x zu beweisen, genügt es also, als *Induktionsanfang* die Richtigkeit der Aussage für atomare Ausdrücke direkt zu beweisen und anschließend als *Induktionsschritt* die Richtigkeit der Aussage für einen zusammengesetzten Ausdruck unter der *Induktionsvoraussetzung* zu beweisen, daß die Aussage für seine Teilausdrücke bereits bewiesen ist. Für diesen Induktionsschritt ist i. d. R. eine Fallunterscheidung nach der Struktur des Ausdrucks, d. h. nach seinem „Hauptoperator“ erforderlich.

3.4.4 Vollständige und partielle Worte von Ausdrücken

3.4.4.1 Lemma

Für zwei Mengen $A, B \subseteq \Sigma$ gilt offensichtlich:

$$A^* \otimes B^* = (A \cup B)^* \quad \text{und} \quad A^* \cap B^* = (A \cap B)^*.$$

3.4.4.2 Satz

1. Für beliebige Interaktionsausdrücke x gelten die Inklusionen

   $$\{\langle\rangle\} \cup \Phi(x) \subseteq \Psi(x) \subseteq \alpha'(x)^* \quad \text{mit} \quad \alpha'(x) = \alpha(x) \cap \Sigma,$$

   d. h. das leere Wort und jedes vollständige Wort von x ist auch ein partielles Wort von x, und sämtliche Worte von x enthalten nur konkrete Aktionen aus dem Alphabet von x.

2. Die Menge $\Psi(x)$ ist *abgeschlossen bzgl. Präfixbildung*, d. h. mit einem Wort $w \in \Psi(x)$ gehören auch alle Präfixe von w zur Menge $\Psi(x)$.

Beweis

1. Je nach Kategorie des Ausdrucks x ergeben sich die Inklusionen wie folgt:

   a) Für einen *atomaren Ausdruck* $x \equiv a$ folgen sie unmittelbar aus der Definition der Mengen $\Phi(x)$, $\Psi(x)$ und $\alpha(x)$.

   b) Für einen *zusammengesetzten Ausdruck* x, der weder eine binäre Synchronisation noch einen Synchronisations-Quantorausdruck darstellt, werden zur Definition der Menge $\Psi(x)$ nur Mengen $\Psi(\xi)$ und ggf. $\Phi(\xi)$ für irgendwelche (ggf. konkretisierten) Teilausdrücke ξ von x verwendet, für die nach Induktionsvoraussetzung und Definition von $\alpha(x)$ gilt:

   $$\Phi(\xi) \subseteq \Psi(\xi) \subseteq \alpha'(\xi)^* \subseteq \alpha'(x)^*,$$

   d. h. sowohl $\Psi(\xi)$ als auch $\Phi(\xi)$ sind Teilmengen von $\alpha'(x)^*$. Aufgrund der Abgeschlossenheit aller verwendeten Mengenoperationen (vgl. § 3.4.2.1) folgt daraus die Inklusion $\Psi(x) \subseteq \alpha'(x)^*$.

   c) Für eine *binäre Synchronisation* ergibt sich diese Aussage wie folgt:[10]

   $$\Psi(y \circ z) = \Psi(y) \otimes \kappa_x(y)^* \cap \Psi(z) \otimes \kappa_x(z)^*$$

   ⇓ Induktionsvoraussetzung.

   $$\subseteq \alpha'(y)^* \otimes \kappa_x(y)^* \cap \alpha'(z)^* \otimes \kappa_x(z)^*$$

   ⇓ Lemma.

   $$= \big(\alpha'(y) \cup \kappa_x(y)\big)^* \cap \big(\alpha'(z) \cup \kappa_x(z)\big)^*$$

   ⇓ Lemma.

   $$= \Big(\big(\alpha'(y) \cup \kappa_x(y)\big) \cap \big(\alpha'(z) \cup \kappa_x(z)\big)\Big)^*$$

   ⇓ Distributivgesetz für $\cup$ und $\cap$.

[10] Im folgenden werden Übergänge von einer Beweiszeile zur nächsten gelegentlich durch eingeschobene ⇓-Zeilen erläutert.

$$= \big(\big(\alpha'(y) \cap \alpha'(z)\big) \cup \big(\alpha'(y) \cap \kappa_x(z)\big) \cup \big(\kappa_x(y) \cap \alpha'(z)\big) \cup \big(\kappa_x(z) \cap \kappa_x(y)\big)\big)^*$$

⇓ Beachte $\alpha'(\xi) \cap \ldots \subseteq \alpha'(\xi) \subseteq \alpha'(x)$
⇓ sowie $\kappa_x(z) \cap \kappa_x(y) = \emptyset$ gemäß Definition von κ_x.

$$\subseteq \alpha'(x)^*.$$

d) Für einen *Synchronisations-Quantorausdruck* ergibt sich die Behauptung analog, wenn man die verwendeten Hilfsmittel (Lemma und Distributivgesetz) geeignet auf unendlich viele Mengen verallgemeinert, was problemlos möglich ist.

e) Die Inklusionen $\Phi(x) \subseteq \Psi(x)$ und $\langle\rangle \in \Psi(x)$ folgen leicht aus den Definitionen von $\Phi(x)$ und $\Psi(x)$ unter Verwendung der Induktionsvoraussetzung und ggf. der Monotonie der verwendeten Mengenoperationen (vgl. § 3.4.2.1).

2. Für die Abgeschlossenheit von $\Psi(x)$ bzgl. Präfixbildung genügt es offensichtlich, die Implikation

$$w\,\langle a\rangle \in \Psi(x) \;\Rightarrow\; w \in \Psi(x)$$

für beliebige Worte $w \in \Sigma^*$ und Aktionen $a \in \Sigma$ zu zeigen, was mit Hilfe der Definitionen und des Induktionsprinzips ebenfalls leicht gelingt.

3.4.5 Vergleich von Operatoren

3.4.5.1 Lemma

Für die Mengenoperationen Konkatenation und Verschränkung sowie sequentielle und parallele Hülle gelten die folgenden Inklusionsbeziehungen:

$$U\,V \subseteq U \otimes V \quad \text{und} \quad U \cup \{\langle\rangle\} \subseteq U^* \subseteq U\# \quad \text{für} \quad U, V \subseteq \Sigma^*.$$

Beweis

Offensichtlich gilt $u\,v \in u \otimes v$, woraus nacheinander die Aussagen $U\,V \subseteq U \otimes V$, $U^n \subseteq \bigotimes^n U$ und somit $U^* \subseteq U\#$ folgen. Die Behauptung $U \cup \{\langle\rangle\} \subseteq U^*$ folgt direkt aus der Definition der sequentiellen Hülle.

3.4.5.2 Satz

1. Eine Konjunktion akzeptiert höchstens, eine Disjunktion mindestens die Worte, die ihre beiden Teilausdrücke akzeptieren:

$$y \bullet z \subseteq y \subseteq y \bigcirc z;$$

$$y \bullet z \subseteq z \subseteq y \bigcirc z.$$

2. Eine Synchronisation ist restriktiver als einer ihrer Teilausdrücke, wenn dessen Alphabet das Alphabet des anderen Teilausdrucks umfaßt:

$$y \circ z \le y \quad \text{für} \quad \alpha(z) \subseteq \alpha(y);$$

$$y \circ z \le z \quad \text{für} \quad \alpha(y) \subseteq \alpha(z).$$

3. Eine Konjunktion ist restriktiver als eine Synchronisation. Wenn die Alphabete ihrer Teilausdrücke gleich sind, ist die Konjunktion äquivalent zur Synchronisation:

$$y \bullet z \le y \circ z;$$

$$y \bullet z = y \circ z \quad \text{für} \quad \alpha(y) = \alpha(z).$$

4. Wenn die Alphabete ihrer Teilausdrücke disjunkt sind, ist die Synchronisation äquivalent zur parallelen Komposition:

$$y \mathbin{\boldsymbol{\circ}} z \;=\; y \circledcirc z \qquad \text{für} \qquad \alpha(y) \cap \alpha(z) = \emptyset.$$

5. Die sequentielle Komposition ist restriktiver als die parallele Komposition. Ebenso ist die sequentielle Iteration restriktiver als die parallele Iteration, beide jedoch weniger restriktiv als die Option und ihr Rumpf:

$$y - z \;\le\; y \circledcirc z;$$

$$y \;\le\; \smile y \;\le\; \ominus y \;\le\; \circledcirc\!\!\!\bigcirc\, y.$$

Beweis

1. Die Aussagen über Konjunktion und Disjunktion folgen unmittelbar aus den Definitionen der Operatoren.

2. Aus $\alpha(z) \subseteq \alpha(y)$ folgt zunächst $\alpha(x) = \alpha(y) \cup \alpha(z) = \alpha(y)$ sowie $\kappa_x(y) = \alpha(x) \setminus \alpha(y) = \emptyset$, d. h. $\kappa_x(y)^* = \{ \langle\rangle \}$. Daher gilt:

$$\Phi(y \mathbin{\boldsymbol{\circ}} z) = \Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^* \subseteq \Phi(y) \otimes \{ \langle\rangle \} = \Phi(y),$$

und analog:

$$\Psi(y \mathbin{\boldsymbol{\circ}} z) \subseteq \Psi(y).$$

Daher gilt $y \mathbin{\boldsymbol{\circ}} z \;\le\; y$. Die „symmetrische“ Aussage $\alpha(y) \subseteq \alpha(z) \;\Rightarrow\; y \mathbin{\boldsymbol{\circ}} z \;\le\; z$ erhält man analog oder durch Ausnutzen der Kommutativität (vgl. § 3.4.7).

3. Für die Beziehung zwischen Konjunktion und Synchronisation gilt grundsätzlich die Inklusion

$$\Phi(y \bullet z) = \Phi(y) \cap \Phi(z) \subseteq \Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^* = \Phi(y \mathbin{\boldsymbol{\circ}} z),$$

da die Mengen $\kappa_x(y)^*$ und $\kappa_x(z)^*$ zumindest das leere Wort enthalten. Unter der Bedingung $\alpha(y) = \alpha(z)$ gilt außerdem $\kappa_x(y) = \kappa_x(z) = \emptyset$, d. h. in diesem Fall enthalten die Mengen $\kappa_x(y)^*$ und $\kappa_x(z)^*$ *nur* das leere Wort. Daraus folgt die Gleichheit von $\Phi(y \bullet z)$ und $\Phi(y \mathbin{\boldsymbol{\circ}} z)$.
Die Aussagen für Ψ ergeben sich vollkommen analog.

4. Für $\alpha(y) \cap \alpha(z) = \emptyset$ gilt:

$$\kappa_x(y) = \alpha(x) \setminus \alpha(y) = \alpha(z) \qquad \text{und} \qquad \kappa_x(z) = \alpha(x) \setminus \alpha(z) = \alpha(y),$$

und somit:

$$\Phi(y \mathbin{\boldsymbol{\circ}} z) = \Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^* = \Phi(y) \otimes \alpha(z)^* \cap \Phi(z) \otimes \alpha(y)^*.$$

Für ein Wort $w \in \Phi(y \mathbin{\boldsymbol{\circ}} z)$ gibt es daher einerseits eine Zerlegung in Teilworte

$$u_1 \in \Phi(y) \subseteq \alpha(y)^* \qquad \text{und} \qquad v_1 \in \alpha(z)^* \qquad \text{mit} \qquad w \in u_1 \otimes v_1$$

und andererseits eine Zerlegung in Teilworte

$$v_2 \in \Phi(z) \subseteq \alpha(z)^* \qquad \text{und} \qquad u_2 \in \alpha(y)^* \qquad \text{mit} \qquad w \in v_2 \otimes u_2 = u_2 \otimes v_2.$$

Wegen $\alpha(y) \cap \alpha(z) = \emptyset$ muß eine Zerlegung von w in Teilworte $u \in \alpha(y)^*$ und $v \in \alpha(z)^*$ mit $w \in u \otimes v$ jedoch *eindeutig* sein, d. h. es muß $u = u_2 = u_1 \in \Phi(y)$ und $v = v_1 = v_2 \in \Phi(z)$ und somit $w \in \Phi(y) \otimes \Phi(z) = \Phi(y \circledcirc z)$ gelten.
Die umgekehrte Inklusion $\Phi(y \circledcirc z) \subseteq \Phi(y \mathbin{\boldsymbol{\circ}} z)$ ergibt sich wie folgt mit Hilfe der Monotonie der Verschränkung (vgl. § 3.4.2.1):

$$\Phi(y \circledcirc z) = \Phi(y) \otimes \Phi(z) \cap \Phi(z) \otimes \Phi(y) \subseteq \Phi(y) \otimes \alpha(z)^* \cap \Phi(z) \otimes \alpha(y)^* = \Phi(y \mathbin{\boldsymbol{\circ}} z).$$

Analog ergibt sich auch wieder $\Psi(y \mathbin{\boldsymbol{\circ}} z) = \Psi(y \circledcirc z)$.

5. Für die Beziehung zwischen sequentieller und paralleler Komposition folgt mit Hilfe des Lemmas und der Inklusion $\Phi(y) \subseteq \Psi(y)$ (vgl. § 3.4.4.2):

   $$\Phi(y - z) = \Phi(y)\,\Phi(z) \subseteq \Phi(y) \otimes \Phi(z) = \Phi(y \circledcirc z),$$

   $$\begin{aligned}\Psi(y - z) &= \Psi(y) \cup \Phi(y)\,\Psi(z) \subseteq \Psi(y) \cup \Psi(y)\,\Psi(z) \subseteq \Psi(y) \otimes \Psi(z) \cup \Psi(y) \otimes \Psi(z)\\ &= \Psi(y \circledcirc z).\end{aligned}$$

   In ähnlicher Weise ergeben sich auch die Aussagen über die unären Operatoren.

3.4.6 Singuläre Ausdrücke und parallele Quantoren

3.4.6.1 Definition

Ein Ausdruck x heißt (syntaktisch) *singulär*, wenn er einem der folgenden Muster entspricht:

$$x \equiv \begin{cases} ◌\, y & \text{mit } ◌ \in \{\, ⌣,\ ⊖,\ ⊚ \,\} \text{ und einem beliebigen Teilausdruck } y,\\ y \bigcirc z & \text{mit mindestens einem singulären Teilausdruck } y \text{ oder } z,\\ y\, ◌\, z & \text{mit } ◌ \in \{\, -,\ \circledcirc,\ ⭘,\ ● \,\} \text{ und zwei singulären Teilausdrücken } y \text{ und } z,\\ \underset{p}{◯}\, y & \text{mit } ◌ \in \{\, \bigcirc,\ \circledcirc,\ ⭘,\ ● \,\} \text{ und einem singulären Teilausdruck } y. \end{cases}$$

3.4.6.2 Satz

Ein Ausdruck x ist genau dann singulär, wenn das leere Wort ein vollständiges Wort von x darstellt, d. h. wenn $\langle\rangle \in \Phi(x)$ gilt.

Beweis

1. Ein atomarer Ausdruck $x \equiv a$ ist nach Definition weder singulär, noch enthält seine Menge $\Phi(x) = \{\,\langle a\rangle\,\} \cap \Sigma^*$ das leere Wort.

2. Ein unärer Ausdruck $x \equiv ◌\, y$ ist nach Definition singulär, und jede der Mengen

   $\Phi(x) = \Phi(y) \cup \{\,\langle\rangle\,\}$ (Option) bzw.

   $\Phi(x) = \Phi(y)^*$ (sequentielle Iteration) bzw.

   $\Phi(x) = \Phi(y)^\#$ (parallele Iteration)

   enthält das leere Wort.

3. Eine Disjunktion $x \equiv y \bigcirc z$ ist genau dann singulär, wenn mindestens einer der Teilausdrücke y oder z singulär ist, d. h. wenn $\langle\rangle \in \Phi(y)$ oder $\langle\rangle \in \Phi(z)$ gilt. Dies ist äquivalent zu $\langle\rangle \in \Phi(y) \cup \Phi(z) = \Phi(x)$.

4. Ein binärer Ausdruck $x \equiv y\, ◌\, z$ mit $◌ \in \{\, -,\ \circledcirc,\ ⭘,\ ● \,\}$ ist genau dann singulär, wenn die Teilausdrücke y und z beide singulär sind, d. h. wenn $\langle\rangle \in \Phi(y)$ und $\langle\rangle \in \Phi(z)$ gilt. Genau dann enthält die Menge

   $\Phi(x) = \Phi(y)\,\Phi(z)$ (sequentielle Komposition) bzw.

   $\Phi(x) = \Phi(y) \otimes \Phi(z)$ (parallele Komposition) bzw.

   $\Phi(x) = \Phi(y) \cap \Phi(z)$ (Konjunktion) bzw.

   $\Phi(x) = \Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^*$ (Synchronisation)

   aber auch das leere Wort.

5. Ein Quantorausdruck $x \equiv \bigcirc_{p} y$ ist genau dann singulär, wenn der Teilausdruck y singulär ist. Da die Singularität eines Ausdrucks nach Definition eine rein syntaktische Eigenschaft ist und die konkretisierten Ausdrücke y_p^ω für alle $\omega \in \Omega$ dieselbe syntaktische Struktur wie der abstrakte Ausdruck y besitzen, folgt aus der Singularität von y die Singularität aller y_p^ω (und somit $\langle\rangle \in \Phi\left(y_p^\omega\right)$ für alle $\omega \in \Omega$) und umgekehrt aus der Nicht-Singularität von y auch die Nicht-Singularität aller y_p^ω (und somit $\langle\rangle \notin \Phi\left(y_p^\omega\right)$ für alle $\omega \in \Omega$). Außerdem folgt aus den Definitionen von $\Phi(x)$, daß $\Phi(x)$ das leere Wort enthält, wenn es in allen $\Phi\left(y_p^\omega\right)$ enthalten ist, und umgekehrt, daß $\Phi(x)$ das leere Wort nicht enthält, wenn es in keinem $\Phi\left(y_p^\omega\right)$ enthalten ist. Zusammengefaßt folgt daraus die Behauptung, daß x genau dann singulär ist, wenn $\Phi(x)$ das leere Wort enthält.

3.4.6.3 Korollar

Ein paralleler Quantorausdruck $x \equiv \circledcirc_{p} y$ ist genau dann *erfüllbar*, d. h. die Menge $\Phi(x)$ seiner vollständigen Worte ist *nicht leer*, wenn sein Rumpf y singulär ist.

Beweis

Wenn y singulär ist, ist auch x singulär, und somit enthält $\Phi(x)$ zumindest das leere Wort.

Wenn y nicht singulär ist, enthält keine der Mengen $\Phi\left(y_p^\omega\right)$ das leere Wort, und somit gilt nach Definition:

$$\Phi(x) = \bigotimes_{\omega \in \Omega} \Phi\left(y_p^\omega\right) = \varnothing.$$

3.4.7 Eigenschaften binärer Operatoren

3.4.7.1 Lemma

1. Die Mengenoperationen Vereinigung und Durchschnitt sowie Konkatenation und Verschränkung sind *assoziativ* und, mit Ausnahme der Konkatenation, auch *kommutativ*. Vereinigung und Durchschnitt sind außerdem *idempotent*. Es gilt also für $U, V, W \subseteq \Sigma^*$:

$$(U \cup V) \cup W = U \cup (V \cup W), \quad U \cup V = V \cup U, \quad U \cup U = U;$$

$$(U \cap V) \cap W = U \cap (V \cap W), \quad U \cap V = V \cap U, \quad U \cap U = U;$$

$$(U\,V)\,W = U\,(V\,W);$$

$$(U \otimes V) \otimes W = U \otimes (V \otimes W), \quad U \otimes V = V \otimes U.$$

2. Konkatenation und Verschränkung sind *distributiv* bzgl. Vereinigung, d. h. es gilt für U, U_1, U_2, V, V_1, $V_2 \subseteq \Sigma^*$:

$$U\left(V_1 \cup V_2\right) = U\,V_1 \cup U\,V_2, \quad \left(U_1 \cup U_2\right) V = U_1\,V \cup U_2\,V;$$

$$U \otimes \left(V_1 \cup V_2\right) = U \otimes V_1 \cup U \otimes V_2, \quad \left(U_1 \cup U_2\right) \otimes V = U_1 \otimes V \cup U_2 \otimes V$$

und entsprechend auch für unendliche Vereinigungen.

3. Die Verschränkung ist *eingeschränkt distributiv* bzgl. Durchschnitt, d. h. für $A, B \subseteq \Sigma$ mit $A \cap B = \varnothing$ und $U_1, U_2 \subseteq A^*$ gilt:

$$\left(U_1 \cap U_2\right) \otimes B^* = U_1 \otimes B^* \cap U_2 \otimes B^*$$

und entsprechend auch für unendliche Durchschnitte.

Beweis

Auch diese Aussagen (mit Ausnahme der letzten) gehören entweder zum mathematischen Allgemeinwissen oder folgen unmittelbar aus den Definitionen der Operationen.

Zu 3: Die Inklusion l.S. $\subseteq$ r.S. folgt unmittelbar aus den Definitionen. Für ein Wort $w \in$ r.S. gibt es wegen $A \cap B = \emptyset$ eine *eindeutige* Zerlegung von w in Teilworte $u \in A^*$ und $v \in B^*$ mit $w \in u \otimes v$. Wegen $w \in U_1 \otimes B^*$ und $w \in U_2 \otimes B^*$ muß das Wort u sowohl in U_1 als auch in U_2 enthalten sein, woraus die Inklusion r.S. $\subseteq$ l.S. folgt.

3.4.7.2 Satz

Alle binären Operatoren von Interaktionsausdrücken sind assoziativ und, mit Ausnahme der sequentiellen Komposition, auch kommutativ, d. h. es gilt:

$(x \bigcirc y) \bigcirc z = x \bigcirc (y \bigcirc z)$ für $\bigcirc \in \{ -, \circ, \circledcirc, \boldsymbol{\circ}, \bullet \}$,

$y \bigcirc z = z \bigcirc y$ für $\bigcirc \in \{ \circ, \circledcirc, \boldsymbol{\circ}, \bullet \}$.

Die Booleschen Operatoren einschließlich der Synchronisation sind außerdem idempotent, d. h. es gilt:

$y \bigcirc y = y$ für $\bigcirc \in \{ \circ, \boldsymbol{\circ}, \bullet \}$.

Beweis

Da die Alphabete der zu vergleichenden Ausdrücke jeweils gleich sind, folgen die meisten Behauptungen unmittelbar aus der Assoziativität und ggf. Kommutativität und Idempotenz der Mengenoperationen Vereinigung und Durchschnitt sowie Konkatenation und Verschränkung (vgl. Lemma).

Für die Assoziativität der sequentiellen Komposition wird außerdem die Distributivität der Konkatenation bzgl. Vereinigung benötigt:

$$\Psi((x - y) - z) = \Psi(x - y) \cup \Phi(x - y)\,\Psi(z) = \Psi(x) \cup \Phi(x)\,\Psi(y) \cup \Phi(x)\,\Phi(y)\,\Psi(z)$$

$$= \Psi(x) \cup \Phi(x)\,(\Psi(y) \cup \Phi(y)\,\Psi(z)) = \Psi(x) \cup \Phi(x)\,\Psi(y - z) = \Psi(x - (y - z)).$$

Für die Assoziativität der Synchronisation ist zu beachten, daß der Teilausdruck $xy \equiv x \boldsymbol{\circ} y$ beispielsweise ein anderes Alphabet besitzt als der Gesamtausdruck $xyz \equiv xy \boldsymbol{\circ} z \equiv (x \boldsymbol{\circ} y) \boldsymbol{\circ} z$.[11] Mit diesen beiden Abkürzungen gilt:

$$\Phi((x \boldsymbol{\circ} y) \boldsymbol{\circ} z)$$

⇓ Definition der Synchronisation

$$= \Phi(x \boldsymbol{\circ} y) \otimes \kappa_{xyz}(xy)^* \;\cap\; \Phi(z) \otimes \kappa_{xyz}(z)^*$$

⇓ Definition der Synchronisation

$$= \left(\Phi(x) \otimes \kappa_{xy}(x)^* \;\cap\; \Phi(y) \otimes \kappa_{xy}(y)^*\right) \otimes \kappa_{xyz}(xy)^* \;\cap\; \Phi(z) \otimes \kappa_{xyz}(z)^*$$

⇓ Distributivität von $\otimes$ bzgl. $\cap$

$$= \Phi(x) \otimes \kappa_{xy}(x)^* \otimes \kappa_{xyz}(xy)^* \;\cap\; \Phi(y) \otimes \kappa_{xy}(y)^* \otimes \kappa_{xyz}(xy)^* \;\cap\; \Phi(z) \otimes \kappa_{xyz}(z)^*$$

⇓ $A^* \otimes B^* = (A \cup B)^*$

[11] xy bezeichnet nicht etwa die Konkatenation oder Sequenz $x - y$, sondern stellt eine abkürzende Bezeichnung für den Ausdruck $x \boldsymbol{\circ} y$ dar. Entsprechendes gilt für den Bezeichner xyz.

$= \Phi(x) \otimes \left(\kappa_{xy}(x) \cup \kappa_{xyz}(xy)\right)^* \;\cap\; \Phi(y) \otimes \left(\kappa_{xy}(y) \cup \kappa_{xyz}(xy)\right)^* \;\cap\; \Phi(z) \otimes \kappa_{xyz}(z)^*$

⇓ Erläuterung folgt

$= \Phi(x) \otimes \kappa_{xyz}(x)^* \;\cap\; \Phi(y) \otimes \kappa_{xyz}(y)^* \;\cap\; \Phi(z) \otimes \kappa_{xyz}(z)^*$,

denn:

$\kappa_{xy}(x) \cup \kappa_{xyz}(xy)$

⇓ Definition von κ

$= (\alpha(xy) \setminus \alpha(x)) \cup (\alpha(xyz) \setminus \alpha(xy))$

$= (\alpha(y) \setminus \alpha(x)) \cup (\alpha(z) \setminus \alpha(x) \setminus \alpha(y))$

⇓ $(A \setminus C) \cup (B \setminus C) = (A \cup B) \setminus C$

$= (\alpha(y) \cup (\alpha(z) \setminus \alpha(y))) \setminus \alpha(x)$

$= (\alpha(y) \cup \alpha(z)) \setminus \alpha(x)$

⇓ Definition von κ

$= \kappa_{xyz}(x)$

und analog:

$\kappa_{xy}(y) \cup \kappa_{xyz}(xy) = \kappa_{xyz}(y)$.

Durch Ausnutzen der Kommutativität und geeignetes „Umbenennen“ der Teilausdrücke erhält man für $\Phi(x \circ (y \circ z))$ genau dasselbe Resultat:

$\Phi(x \circ (y \circ z)) = \Phi((y \circ z) \circ x)$

$= \Phi(y) \otimes \kappa_{xyz}(y)^* \;\cap\; \Phi(z) \otimes \kappa_{xyz}(z)^* \;\cap\; \Phi(x) \otimes \kappa_{xyz}(x)^*$,

das heißt

$\Phi((x \circ y) \circ z) = \Phi(x \circ (y \circ z))$.

Ersetzt man schließlich in allen obigen Formeln Φ durch Ψ, so erhält man die entsprechende Aussage auch für die Menge der partiellen Worte:

$\Psi((x \circ y) \circ z) = \Psi(x \circ (y \circ z))$.

3.4.8 Eigenschaften unärer Operatoren

3.4.8.1 Lemma

Die Mengenoperationen sequentielle und parallele Hülle sind *idempotent*, d. h. es gilt:

$U^{**} = U^*$ und $U\#\# = U\#$.

Beweis

Aufgrund der Distributivität der Konkatenation (vgl. § 3.4.7.1) gilt zunächst:

$$(U^*)\,(U^*) = \left(\bigcup_{m=0}^{\infty} U^m\right)\left(\bigcup_{n=0}^{\infty} U^n\right) = \bigcup_{m=0}^{\infty}\bigcup_{n=0}^{\infty} U^m\, U^n = \bigcup_{m,n=0}^{\infty} U^{m+n} = \bigcup_{k=0}^{\infty} U^k = U^*,$$

was sich mit Hilfe vollständiger Induktion leicht zu

$(U^*)^n = U^*$ für $n \geq 1$

verallgemeinern läßt. Daher gilt:

$$U^{**} = (U^*)^0 \cup \bigcup_{n=1}^{\infty} (U^*)^n = \{ \langle\rangle \} \cup \bigcup_{n=1}^{\infty} U^* = \{ \langle\rangle \} \cup U^* = U^*.$$

Die Idempotenz der parallelen Hülle ergibt sich analog.

3.4.8.2 Satz

Alle unären Operatoren von Interaktionsausdrücken sind idempotent, d. h. es gilt:

$$\bigcirc \bigcirc y = \bigcirc y \qquad \text{für } \bigcirc \in \{ \circlearrowright, \ominus, \circledcirc \}.$$

Außerdem nimmt die „Potenz“ der Operatoren in der Reihenfolge $\circlearrowright$, $\ominus$, $\circledcirc$ zu, d. h. es gilt:

$$\ominus \circlearrowright y = \circlearrowright \ominus y = \ominus y,$$

$$\circledcirc \circlearrowright y = \circlearrowright \circledcirc y = \circledcirc \ominus y = \ominus \circledcirc y = \circledcirc y.$$

Beweis

1. Die Idempotenz folgt leicht aus den Definitionen und den Idempotenz-Eigenschaften der Hüllenoperationen (vgl. Lemma).
2. Die übrigen Behauptungen können unter Verwendung bereits bewiesener Aussagen wie folgt hergeleitet werden. Gemäß § 3.4.5.2 gilt zunächst:

   $$y \le \circlearrowright y \le \ominus y.$$

   Aufgrund der Monotonie (vgl. § 3.4.2.3) und der Idempotenz der sequentiellen Iteration folgt daraus die Ungleichungskette:

   $$\ominus y \le \ominus \circlearrowright y \le \ominus \ominus y = \ominus y,$$

   die gemäß § 3.4.2.3 die Gleichung $\ominus \circlearrowright y = \ominus y$ impliziert.
3. Wendet man denselben Satz (§ 3.4.5.2) auf den Ausdruck $(\ominus y)$ an, so erhält man die Ungleichungskette

   $$(\ominus y) \le \circlearrowright (\ominus y) \le \ominus (\ominus y) = \ominus y,$$

   aus der die Gleichung $\circlearrowright \ominus y = \ominus y$ folgt.
4. Nach demselben Prinzip lassen sich auch die verbleibenden Behauptungen herleiten.

3.4.9 Eigenschaften von Quantoren

3.4.9.1 Satz

Quantoren können sowohl mit gleichartigen Quantoren als auch mit gleichartigen binären Operatoren vertauscht werden, d. h. es gilt für $\bigcirc \in \{ \circ, \circledcirc, \boldsymbol{\circ}, \bullet \}$:

$$\underset{p}{\bigcirc} \underset{q}{\bigcirc} y = \underset{q}{\bigcirc} \underset{p}{\bigcirc} y,$$

$$\underset{p}{\bigcirc} (y \circ z) = \left(\underset{p}{\bigcirc} y \right) \circ \left(\underset{p}{\bigcirc} z \right).$$

3.4.9.2 Beweis

Disjunktion und Konjunktion

Für die Booleschen Operatoren $\bigcirc$ und $\bullet$ folgen die Behauptungen unmittelbar aus der Kommutativität der Mengenoperationen Vereinigung und Durchschnitt.

Parallele Komposition

Partielle Worte

Für ein Wort $w \in \Sigma^*$ gilt die folgende Kette von Implikationen:

$$w \in \Psi\Big(\circledcirc_p \circledcirc_q y\Big) = \bigotimes_{\omega\in\Omega} \Psi\Big(\Big(\circledcirc_q y\Big)_p^{\omega}\Big) = \bigotimes_{\omega\in\Omega} \Psi\Big(\circledcirc_q y_p^{\omega}\Big) = \bigcup_{\substack{m\in\mathbb{N}\\ \omega_1\neq\ldots\neq\omega_m\in\Omega}} \bigotimes_{i=1}^{m} \Psi\Big(\circledcirc_q y_p^{\omega_i}\Big)$$

$$\Rightarrow \quad \exists\, m \in \mathbb{N},\ \omega_1 \neq \ldots \neq \omega_m \in \Omega : w \in \bigotimes_{i=1}^{m} \Psi\Big(\circledcirc_q y_p^{\omega_i}\Big)$$

$$\Rightarrow \quad \exists\, m \in \mathbb{N},\ \omega_1 \neq \ldots \neq \omega_m \in \Omega,\ w_1, \ldots, w_m \in \Sigma^* : w \in \bigotimes_{i=1}^{m} w_i,$$

$$w_i \in \Psi\Big(\circledcirc_q y_p^{\omega_i}\Big) = \bigotimes_{\pi\in\Omega} \Psi\big(y_{p,q}^{\omega_i,\pi}\big) = \bigcup_{\substack{n\in\mathbb{N}\\ \pi_1\neq\ldots\neq\pi_n\in\Omega}} \bigotimes_{j=1}^{n} \Psi\big(y_{p,q}^{\omega_i,\pi_j}\big) \quad \text{für } i = 1, \ldots, m$$

$$\Rightarrow \quad \exists\, m \in \mathbb{N},\ \omega_1 \neq \ldots \neq \omega_m \in \Omega,\ w_1, \ldots, w_m \in \Sigma^*,$$
$$n_1, \ldots, n_m \in \mathbb{N},\ \pi_{11} \neq \ldots \neq \pi_{1n_1} \in \Omega, \ldots\ldots, \pi_{m1} \neq \ldots \neq \pi_{mn_m} \in \Omega :$$
$$w \in \bigotimes_{i=1}^{m} w_i, \quad w_i \in \bigotimes_{j=1}^{n_i} \Psi\big(y_{p,q}^{\omega_i,\pi_{ij}}\big) \quad \text{für } i = 1, \ldots, m$$

$\Downarrow$ Wähle $\{\pi_1, \ldots, \pi_n\} = \{\pi_{11}, \ldots, \pi_{mn_m}\}$ und beachte, daß alle Mengen $\Psi\big(y_{p,q}^{\omega,\pi}\big)$ das leere
$\Downarrow$ Wort enthalten; daraus folgt $\bigotimes_{j=1}^{n_i} \Psi\big(y_{p,q}^{\omega_i,\pi_{ij}}\big) \subseteq \bigotimes_{j=1}^{n} \Psi\big(y_{p,q}^{\omega_i,\pi_j}\big)$ für $i = 1, \ldots, m$.

$$\Rightarrow \quad \exists\, m \in \mathbb{N},\ \omega_1 \neq \ldots \neq \omega_m \in \Omega,\ w_1, \ldots, w_m \in \Sigma^*,$$
$$n \in \mathbb{N},\ \pi_1 \neq \ldots \neq \pi_n \in \Omega :$$
$$w \in \bigotimes_{i=1}^{m} w_i, \quad w_i \in \bigotimes_{j=1}^{n} \Psi\big(y_{p,q}^{\omega_i,\pi_j}\big) \quad \text{für } i = 1, \ldots, m$$

$\Downarrow$ Kommutativität endlicher Verschränkungen

$$\Rightarrow \quad \exists\, n \in \mathbb{N},\ \pi_1 \neq \ldots \neq \pi_n \in \Omega,$$
$$m \in \mathbb{N},\ \omega_1 \neq \ldots \neq \omega_m \in \Omega :$$
$$w \in \bigotimes_{i=1}^{m} \bigotimes_{j=1}^{n} \Psi\big(y_{p,q}^{\omega_i,\pi_j}\big) = \bigotimes_{j=1}^{n} \bigotimes_{i=1}^{m} \Psi\big(y_{p,q}^{\omega_i,\pi_j}\big)$$

$$\Rightarrow \quad \exists\, n \in \mathbb{N},\ \pi_1 \neq \ldots \neq \pi_n \in \Omega,\ w'_1, \ldots, w'_n \in \Sigma^*,$$
$$m \in \mathbb{N},\ \omega_1 \neq \ldots \neq \omega_m \in \Omega :$$
$$w \in \bigotimes_{j=1}^{n} w'_j, \quad w'_j \in \bigotimes_{i=1}^{m} \Psi\big(y_{p,q}^{\omega_i,\pi_j}\big) \quad \text{für } j = 1, \ldots, n$$

$$\Rightarrow \quad \exists\, n \in \mathbb{N},\ \pi_1 \neq \ldots \neq \pi_n \in \Omega,\ w'_1, \ldots, w'_n \in \Sigma^* : w \in \bigotimes_{j=1}^{n} w'_j,$$

$$w'_j \in \bigcup_{\substack{m\in\mathbb{N}\\ \omega_1\neq\ldots\neq\omega_m\in\Omega}} \bigotimes_{i=1}^{m} \Psi\big(y_{p,q}^{\omega_i,\pi_j}\big) = \bigotimes_{\omega\in\Omega} \Psi\big(y_{p,q}^{\omega,\pi_j}\big) = \Psi\Big(\circledcirc_p y_q^{\pi_j}\Big) \quad \text{für } j = 1, \ldots, n$$

$$\Rightarrow \quad \exists\, n \in \mathbb{N}, \pi_1 \neq \ldots \neq \pi_n \in \Omega : w \in \bigotimes_{j=1}^{n} \Psi\left(\circledcirc_p \, y_q^{\pi_j} \right)$$

$$\Rightarrow \quad w \in \bigcup_{\substack{n \in \mathbb{N} \\ \pi_1 \neq \ldots \neq \pi_n \in \Omega}} \bigotimes_{j=1}^{n} \Psi\left(\circledcirc_p \, y_q^{\pi_j} \right) = \bigotimes_{\pi \in \Omega} \Psi\left(\circledcirc_p \, y_q^{\pi} \right) = \bigotimes_{\pi \in \Omega} \Psi\left(\left(\circledcirc_p \, y \right)_q^{\pi} \right) = \Psi\left(\circledcirc_q \circledcirc_p \, y \right).$$

Zusammengefaßt bedeutet dies:

$$w \in \Psi\left(\circledcirc_p \circledcirc_q \, y \right) \Rightarrow w \in \Psi\left(\circledcirc_q \circledcirc_p \, y \right),$$

das heißt:

$$\Psi\left(\circledcirc_p \circledcirc_q \, y \right) \subseteq \Psi\left(\circledcirc_q \circledcirc_p \, y \right).$$

Da diese Inklusion für *beliebige* Quantorparameter $p, q \in \Pi$ gilt, kann man die Rollen von p und q vertauschen, d. h. es gilt ebenso:

$$\Psi\left(\circledcirc_q \circledcirc_p \, y \right) \subseteq \Psi\left(\circledcirc_p \circledcirc_q \, y \right).$$

Faßt man beide Inklusionen zusammen, so erhält man die Gleichung

$$\Psi\left(\circledcirc_q \circledcirc_p \, y \right) = \Psi\left(\circledcirc_p \circledcirc_q \, y \right).$$

Vollständige Worte

Wenn der Ausdruck y nicht singulär ist, ist sowohl die Menge $\Phi\left(\circledcirc_p \circledcirc_q \, y \right)$ als auch die Menge $\Phi\left(\circledcirc_q \circledcirc_p \, y \right)$ leer (vgl. § 3.4.6.3), d. h. die Mengen sind gleich. Andernfalls enthalten alle Mengen $\Phi\left(y_{p,q}^{\omega,\pi} \right)$ das leere Wort, so daß die obige Herleitung in gleicher Weise auch auf die Mengen Φ statt Ψ angewandt werden kann.

Synchronisation

Bei der Betrachtung der Synchronisation ist wieder auf eine korrekte Behandlung der Alphabete zu achten. Mit den abkürzenden Bezeichnungen $y_{\omega\pi} \equiv y_{p,q}^{\omega,\pi}$, $y_\omega \equiv \bullet_q \, y_p^{\omega}$ und $x \equiv \bullet_p \bullet_q \, y$ gilt:

$$\Phi\left(\bullet_p \bullet_q \, y \right)$$

$$= \bigcap_{\omega \in \Omega} \Phi\left(\left(\bullet_q \, y \right)_p^{\omega} \right) \otimes \kappa_x\left(y_\omega \right)^* = \bigcap_{\omega \in \Omega} \Phi\left(\bullet_q \, y_p^{\omega} \right) \otimes \kappa_x\left(y_\omega \right)^*$$

$$= \bigcap_{\omega \in \Omega} \left(\bigcap_{\pi \in \Omega} \Phi\left(y_{p,q}^{\omega,\pi} \right) \otimes \kappa_{y_\omega}\left(y_{\omega\pi} \right)^* \right) \otimes \kappa_x\left(y_\omega \right)^*$$

⇣ Distributivität von ⊗ bzgl. ∩ (vgl. § 3.4.7.1).
⇣ (Die Voraussetzung $U_i \subseteq A^*$ mit $A \cap B = \emptyset$ ist für $A = \alpha\left(y_\omega \right)$ erfüllt.)

$$= \bigcap_{\omega \in \Omega} \bigcap_{\pi \in \Omega} \Phi\left(y_{\omega\pi} \right) \otimes \kappa_{y_\omega}\left(y_{\omega\pi} \right)^* \otimes \kappa_x\left(y_\omega \right)^*$$

⇣ $A^* \otimes B^* = (A \cup B)^*$

$$= \bigcap_{\omega,\, \pi \in \Omega} \Phi\left(y_{\omega\pi}\right) \otimes \left(\kappa_{y_\omega}\left(y_{\omega\pi}\right) \cup \kappa_x\left(y_\omega\right)\right)^*$$

$$\Downarrow\ \kappa_{y_\omega}\left(y_{\omega\pi}\right) \cup \kappa_x\left(y_\omega\right) = \left(\alpha\left(y_\omega\right) \setminus \alpha\left(y_{\omega\pi}\right)\right) \cup \left(\alpha(x) \setminus \alpha\left(y_\omega\right)\right) = \alpha(x) \setminus \alpha\left(y_{\omega\pi}\right) = \kappa_x\left(y_{\omega\pi}\right)$$

$$= \bigcap_{\omega,\, \pi \in \Omega} \Phi\left(y_{\omega\pi}\right) \otimes \kappa_x\left(y_{\omega\pi}\right)^* = \bigcap_{\omega,\, \pi \in \Omega} \Phi\left(y_{p,\, q}^{\omega,\, \pi}\right) \otimes \kappa_x\left(y_{p,\, q}^{\omega,\, \pi}\right)^* =: C.$$

Vertauscht man die Rollen der Parameter p und q, so erhält man entsprechend:

$$\Phi\left(\underset{q}{\boldsymbol{\bigcirc}}\ \underset{p}{\boldsymbol{\bigcirc}}\ y\right)$$

$$= \bigcap_{\omega,\, \pi \in \Omega} \Phi\left(y_{q,\, p}^{\omega,\, \pi}\right) \otimes \kappa_x\left(y_{q,\, p}^{\omega,\, \pi}\right)^*$$

$\Downarrow$ Vertauschung der Rollen von ω und π

$$= \bigcap_{\pi,\, \omega \in \Omega} \Phi\left(y_{q,\, p}^{\pi,\, \omega}\right) \otimes \kappa_x\left(y_{q,\, p}^{\pi,\, \omega}\right)^*$$

$$= \bigcap_{\omega,\, \pi \in \Omega} \Phi\left(y_{p,\, q}^{\omega,\, \pi}\right) \otimes \kappa_x\left(y_{p,\, q}^{\omega,\, \pi}\right)^* = C,$$

und somit:

$$\Phi\left(\underset{p}{\boldsymbol{\bigcirc}}\ \underset{q}{\boldsymbol{\bigcirc}}\ y\right) = \Phi\left(\underset{q}{\boldsymbol{\bigcirc}}\ \underset{p}{\boldsymbol{\bigcirc}}\ y\right).$$

In gleicher Weise erhält man auch die Beziehung für die Menge der partiellen Worte:

$$\Psi\left(\underset{p}{\boldsymbol{\bigcirc}}\ \underset{q}{\boldsymbol{\bigcirc}}\ y\right) = \Psi\left(\underset{q}{\boldsymbol{\bigcirc}}\ \underset{p}{\boldsymbol{\bigcirc}}\ y\right).$$

Vertauschung von Quantoren und binären Operatoren

Die Beweise zur Vertauschung von Quantoren mit gleichartigen binären Operatoren verlaufen prinzipiell ähnlich, wobei die „Komplexität“ der Umformungen etwas geringer ist als bei der Vertauschung zweier Quantoren.

3.4.10 Rekursionsgleichungen der Iterationsoperatoren

3.4.10.1 Satz

Für die Iterationsoperatoren $\ominus$ und $\boldsymbol{\circledcirc}$ gelten die folgenden Rekursionsgleichungen:

$$x = \smile (y - x) \qquad \text{für } x \equiv \ominus\, y,$$

$$x = \smile (y \circledcirc x) \qquad \text{für } x \equiv \boldsymbol{\circledcirc}\, y.$$

Beweis

Für den Ausdruck $\smile (y - \ominus\, y)$ gilt einerseits:

$$\Phi(\smile (y - \ominus\, y)) = \{\,\langle\rangle\,\} \cup \Phi(y)\, \Phi(y)^* = \{\,\langle\rangle\,\} \cup \Phi(y) \bigcup_{n=0}^{\infty} (\Phi(y))^n = (\Phi(y))^0 \cup \bigcup_{n=1}^{\infty} (\Phi(y))^n$$

$$= \bigcup_{n=0}^{\infty} (\Phi(y))^n = \Phi(y)^* = \Phi(\ominus\, y),$$

und andererseits:

$$\Psi(\smile(y - \ominus y)) = \Psi(y) \cup \Phi(y)\,\Phi(y)^*\,\Psi(y)$$

⇓ $\Psi(y)$ ausklammern (Distributivität der Konkatenation bzgl. Vereinigung)

$$= \big(\{\langle\rangle\} \cup \Phi(y)\,\Phi(y)^*\big)\,\Psi(y)$$

⇓ vgl. oben

$$= \Phi(y)^*\,\Psi(y) = \Psi(\ominus y),$$

und somit (da die Alphabete offensichtlich ebenfalls gleich sind):

$$\smile(y - \ominus y) = \ominus y.$$

Analog ergibt sich auch die Gleichung für die parallele Iteration.

3.4.10.2 Anmerkung

Da Makrodefinitionen, wie bereits mehrfach erwähnt, nicht rekursiv sein dürfen, können die genannten Rekursionsgleichungen *nicht* zur Definition der Ausdrücke $x \equiv \ominus y$ bzw. $x \equiv \circledcirc y$ verwendet werden. Sie sind daher primär von theoretischem Interesse.

3.4.11 Redundanz der parallelen Iteration

3.4.11.1 Satz

Die parallele Iteration $\circledcirc y$ ist äquivalent zu dem Quantorausdruck $x \equiv \mathop{\circledcirc}\limits_{p} \smile y$, wobei $p \in \Pi$ ein *anonymer Parameter* ist, der im Ausdruck y nicht vorkommt.

Beweis

Sei $z \equiv \smile y$, d. h. $x \equiv \mathop{\circledcirc}\limits_{p} z$. Da der Parameter p im Ausdruck y nicht vorkommt, gilt $z_p^\omega \equiv \smile y$ für alle $\omega \in \Omega$. Da somit die Mengen

$$\Phi\big(z_p^\omega\big) = \Phi(\smile y) = \Phi(y) \cup \{\langle\rangle\}$$

für alle $\omega \in \Omega$ das leere Wort enthalten, gilt für die Menge $\Phi(x)$ der vollständigen Worte von x:

$$\Phi(x) = \bigotimes_{\omega\in\Omega} \Phi\big(z_p^\omega\big) = \bigcup_{\substack{n\in\mathbb{N}\\ \omega_1\neq\ldots\neq\omega_n\in\Omega}} \bigotimes_{i=1}^{n} \Phi\big(z_p^{\omega_i}\big) = \bigcup_{\substack{n\in\mathbb{N}\\ \omega_1\neq\ldots\neq\omega_n\in\Omega}} \bigotimes^{n} \Phi(\smile y) = \bigcup_{n=1}^{\infty} \bigotimes^{n} \Phi(\smile y)$$

⇓ $\bigotimes^{0} \Phi(\smile y) = \{\langle\rangle\} \subseteq \Phi(y) \cup \{\langle\rangle\} = \Phi(\smile y) = \bigotimes^{1} \Phi(\smile y)$

$$= \bigcup_{n=0}^{\infty} \bigotimes^{n} \Phi(\smile y) = \Phi(\smile y)\# = \Phi(\circledcirc \smile y)$$

⇓ $\circledcirc \smile y = \circledcirc y$ (vgl. § 3.4.8.2)

$$= \Phi(\circledcirc y).$$

Für die Menge $\Psi(x)$ der partiellen Worte von x ergibt sich analog:

$$\Psi(x) = \Psi(\circledcirc y).$$

3.4.11.2 Anmerkung

Streng genommen ist der parallele Iterationsoperator $\circledcirc$ somit redundant. Aufgrund seiner Analogie zum sequentiellen Iterationsoperator $\ominus$ sowie der Tatsache, daß der Ausdruck $\circledcirc y$ die intendierte Se-

mantik wesentlich einfacher und klarer zum Ausdruck bringt als der äquivalente Ersatzausdruck $\underset{p}{\circledcirc} \curvearrowright y$, ist der Operator $\circledcirc$ dennoch ein elementarer Bestandteil von Interaktionsausdrücken.

Für eine praktische Implementierung kann die Äquivalenz $\circledcirc\, y = \underset{p}{\circledcirc} \curvearrowright y$ allerdings dazu verwendet werden, einen Ausdruck $\circledcirc\, y$ „unter der Oberfläche" durch den Ausdruck $\underset{p}{\circledcirc} \curvearrowright y$ zu ersetzen, um so den Aufwand für eine separate Implementierung des Operators $\circledcirc$ einzusparen (vgl. § 4.6.2.4).

3.5 Ausdrucksmächtigkeit von Interaktionsausdrücken

Offensichtlich entsprechen die Operatoren für sequentielle Komposition und Iteration sowie Disjunktion genau den drei Basisoperatoren *regulärer Ausdrücke* [Hopcroft90, Schöning95], wenn auch in einer etwas ungewohnten Notation. Da Interaktionsausdrücke darüber hinaus weitere Operatoren anbieten, liegt die Vermutung nahe, daß sie eine echte *Erweiterung* regulärer Ausdrücke darstellen. Dies wirft grundsätzlich die Frage auf, welche *Ausdrucksmächtigkeit* Interaktionsausdrücke besitzen und an welcher Stelle der *Chomsky-Hierarchie* sie einzuordnen sind. Diese Fragen werden im folgenden etwas näher untersucht.

3.5.1 Definitionen

1. Die *Ausdrucksmächtigkeit* $\mathrm{E}(F)$ eines Formalismus F (wie z. B. reguläre oder Interaktionsausdrücke) kann definiert werden als die *Menge aller Sprachen*, die durch Ausdrücke (o. ä.) dieses Formalismus beschrieben werden können:

   $$\mathrm{E}(F) = \{\, \mathrm{L}(x) \mid x \in F \,\}.$$

   Damit Interaktionsausdrücke in diesem Zusammenhang mit „klassischen" Formalismen wie z. B. regulären Ausdrücken oder kontextfreien Grammatiken verglichen werden können, wird die *Sprache* $\mathrm{L}(x)$ eines Interaktionsausdrucks x als die Menge seiner *vollständigen Worte* definiert (vgl. auch § 3.2.3):

   $$\mathrm{L}(x) = \Phi(x).$$

2. Ein Formalismus F_1 heißt *mindestens so ausdrucksstark* wie ein anderer Formalismus F_2, wenn die Inklusion $\mathrm{E}(F_1) \supseteq \mathrm{E}(F_2)$ gilt; F_1 heißt *echt ausdrucksstärker* als F_2, wenn darüber hinaus $\mathrm{E}(F_1) \neq \mathrm{E}(F_2)$ gilt, d. h. wenn es einen Ausdruck $x \in F_1$ mit $\mathrm{L}(x) \notin \mathrm{E}(F_2)$ gibt.

3. Ein Formalismus F_1 heißt *teilweise ausdrucksstärker* als ein anderer Formalismus F_2, wenn es einen Ausdruck $x \in F_1$ gibt, dessen Sprache $\mathrm{L}(x)$ nicht in der Menge $\mathrm{E}(F_2)$ enthalten ist. Zwei Formalismen F_1 und F_2 heißen *nicht vergleichbar*, wenn F_1 teilweise ausdrucksstärker als F_2 und F_2 teilweise ausdrucksstärker als F_1 ist, d. h. wenn es Ausdrücke $x_1 \in F_1$ mit $\mathrm{L}(x_1) \notin \mathrm{E}(F_2)$ und $x_2 \in F_2$ mit $\mathrm{L}(x_2) \notin \mathrm{E}(F_1)$ gibt.

3.5.2 Vergleich von Interaktionsausdrücken mit regulären Ausdrücken

Reguläre Ausdrücke werden durch Anwendung der Operationen *Konkatenation* (entspricht sequentieller Komposition), *Selektion* (entspricht Disjunktion) und *Hüllenbildung* (entspricht sequentieller Iteration) aus Aktionen bzw. *Symbolen* $a \in \Sigma$ und ggf. den *speziellen Ausdrücken* $\varnothing$ (mit $\mathrm{L}(\varnothing) = \varnothing$) und ε (mit $\mathrm{L}(\varepsilon) = \{\langle\rangle\}$) gebildet [Hopcroft90, Schöning95]. Läßt man die Ausdrücke $\varnothing$ und ε, deren praktische Bedeutung äußerst gering ist, zunächst außer acht, so läßt sich also jeder reguläre Ausdruck auch als Interaktionsausdruck auffassen.

Aber auch die beiden speziellen regulären Ausdrücke $\varnothing$ und ε lassen sie sich wie folgt durch Interaktionsausdrücke *nachbilden*:

1. Für den Interaktionsausdruck $x_1 \equiv a \bullet b$ mit zwei beliebigen, aber *verschiedenen* Aktionen $a, b \in \Sigma$ gilt:

$$\mathrm{L}(x_1) = \Phi(x_1) = \Phi(a \bullet b) = \Phi(a) \cap \Phi(b) = \{\langle a \rangle\} \cap \{\langle b \rangle\} = \emptyset = \mathrm{L}(\emptyset),$$

   d. h. x_1 ist klassisch äquivalent zu dem regulären Ausdruck $\emptyset$.

2. Für den Interaktionsausdruck $x_2 \equiv \smile (a \bullet b) \equiv \smile x_1$ gilt:

$$\mathrm{L}(x_2) = \Phi(x_2) = \Phi(\smile x_1) = \Phi(x_1) \cup \{\langle\rangle\} = \emptyset \cup \{\langle\rangle\} = \{\langle\rangle\} = \mathrm{L}(\varepsilon),$$

   d. h. x_2 ist klassisch äquivalent zu dem regulären Ausdruck ε.

Somit sind Interaktionsausdrücke *mindestens so ausdrucksstark* wie reguläre Ausdrücke. Im folgenden Abschnitt 3.5.3.1 wird außerdem ein Beispiel eines Interaktionsausdrucks angegeben, dessen Sprache *nicht kontextfrei* und somit auch *nicht regulär* ist. Daher sind Interaktionsausdrücke sogar *echt ausdrucksstärker* als reguläre Ausdrücke.

Anmerkung: Da reguläre Ausdrücke bzw. Mengen *abgeschlossen bzgl. Durchschnittsbildung* sind [Hopcroft90, Schöning95], stellt die Hinzunahme der *Konjunktion* keine Erweiterung der Ausdrucksmächtigkeit von regulären Ausdrücken dar. Ebenso läßt sich − durch Konstruktion eines geeigneten *Produktautomaten* [Coleman94, Harel87] − zeigen, daß auch die *parallele Komposition* und die *Synchronisation* theoretisch redundant sind.

Dies bedeutet, daß − abgesehen von parametrisierten Ausdrücken und Quantoren − lediglich die *parallele Iteration* eine echte Erweiterung der Ausdrucksmächtigkeit gegenüber regulären Ausdrücken darstellt, oder anders ausgedrückt: Elementare Interaktionsausdrücke *ohne* parallele Iteration besitzen die gleiche Ausdrucksmächtigkeit wie reguläre Ausdrücke.

3.5.3 Vergleich von Interaktionsausdrücken mit kontextfreien Grammatiken

3.5.3.1 Überlegenheit von Interaktionsausdrücken

Für den Interaktionsausdruck

$$x \equiv (\ominus a - \ominus b - \ominus c) \bullet \circledcirc (a - b - c) \qquad \text{(vgl. Abb. 3.29)}$$

gilt:[12]

$$\begin{aligned}\mathrm{L}(x) = \Phi(x) &= \Phi((\ominus a - \ominus b - \ominus c) \bullet \circledcirc (a - b - c)) \\ &= \Phi(\ominus a - \ominus b - \ominus c) \cap \Phi(\circledcirc (a - b - c)) \\ &= \left\{ \langle a^k b^m c^n \rangle \,\middle|\, k, m, n \in \mathbb{N}_0 \right\} \cap \{ w \in \{a, b, c\}^* \mid w_a = w_b = w_c \in \mathbb{N}_0 \} \\ &= \{ \langle a^n b^n c^n \rangle \mid n \in \mathbb{N}_0 \}.\end{aligned}$$

Diese Sprache stellt ein typisches Beispiel einer *nicht kontextfreien* Sprache dar, wie man durch Anwendung des *Pumping-Lemmas* für kontextfreie Sprachen leicht zeigen kann [Hopcroft90, Schöning95]. Somit sind Interaktionsausdrücke zumindest *teilweise ausdrucksstärker* als kontextfreie

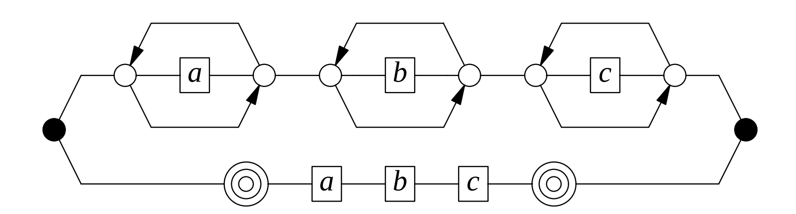


Abbildung 3.29: Ausdruck $(\ominus a - \ominus b - \ominus c) \bullet \circledcirc (a - b - c)$

[12] w_a (bzw. w_b bzw. w_c) bezeichne die Anzahl der Aktionen a (bzw. b bzw. c) im Wort w.

Grammatiken, und es stellt sich die Frage, ob sie möglicherweise sogar echt ausdrucksstärker sind, d. h. ob man *jede* kontextfreie Sprache durch einen äquivalenten Interaktionsausdruck beschreiben kann.

3.5.3.2 Ebenbürtigkeit von Interaktionsausdrücken

Da Interaktionsausdrücken jedoch das für kontextfreie Grammatiken wesentliche Prinzip der *Rekursion* fehlt, erscheint dies auf den ersten Blick eher unwahrscheinlich. Dennoch ist es − zur eigenen Überraschung des Verfassers − z. B. möglich, *geklammerte arithmetische Ausdrücke*, also ein typisches Beispiel einer *rekursiven* kontextfreien Grammatik, mit Hilfe eines Interaktionsausdrucks zu beschreiben:

$$x \equiv [\ominus \mathsf{L} - \mathsf{NUM} - \ominus \mathsf{R} - \ominus (\mathsf{OP} - \ominus \mathsf{L} - \mathsf{NUM} - \ominus \mathsf{R})] \bullet \circledcirc (\mathsf{L} - \mathsf{R}) \qquad \text{(vgl. Abb. 3.30).}$$

Dieser Ausdruck erlaubt Folgen von Zahlen (NUM), Operatoren (OP) und Klammern (L bzw. R), die folgenden Bedingungen genügen:

1. Läßt man Klammern außer acht, müssen sich Zahlen und Operatoren abwechseln, und sowohl am Anfang als auch am Ende eines arithmetischen Ausdrucks muß eine Zahl stehen.
2. Öffnende bzw. schließende Klammern (L bzw. R) dürfen nur vor bzw. nach Zahlen auftreten (Teilausdrücke $\mathsf{L\text{-}NUM\text{-}R} \equiv \ominus \mathsf{L} - \mathsf{NUM} - \ominus \mathsf{R}$).
3. Die Anzahl der öffnenden Klammern muß in jedem Präfix des Ausdrucks mindestens so groß sein wie die Anzahl der schließenden Klammern; im gesamten Ausdruck muß die Anzahl gleich sein (Teilausdruck $\circledcirc (\mathsf{L} - \mathsf{R})$).

Somit beschreibt der Interaktionsausdruck x genau dieselbe Sprache L(x) wie die kontextfreie Grammatik

$$x \rightarrow \mathsf{NUM} \mid x \ \mathsf{OP} \ x \mid \mathsf{L} \ x \ \mathsf{R},$$

bei der die rekursive Formulierung im wesentlichen nur zur Gewährleistung von Bedingung 3, d. h. zum *Zählen* von öffnenden und schließenden Klammern, gebraucht wird.

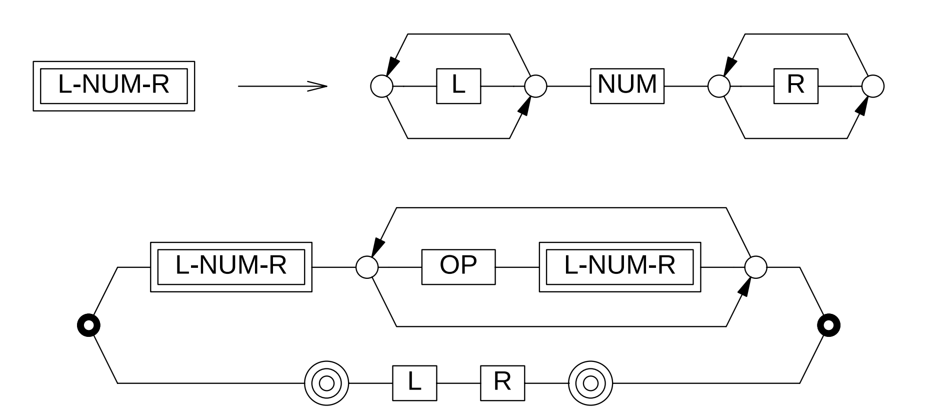


Abbildung 3.30: Interaktionsgraph zur Beschreibung geklammerter arithmetischer Ausdrücke

3.5.3.3 Unterlegenheit von Interaktionsausdrücken?

Anders verhält es sich jedoch, wenn man *zwei verschiedene Arten* von Klammern (z. B. L_1 und R_1 sowie L_2 und R_2) zur Gruppierung von Ausdrücken verwenden möchte, die jedoch *konsistent* verwendet werden müssen („Klammerproblem"). Die zugehörige kontextfreie Grammatik

$x \to \mathsf{NUM} \mid x\ \mathsf{OP}\ x \mid \mathsf{L}_1\ x\ \mathsf{R}_1 \mid \mathsf{L}_2\ x\ \mathsf{R}_2$

unterscheidet sich rein äußerlich nur geringfügig von der vorigen. Allerdings wird die rekursive Formulierung (bzw. der *Stack* eines Push-down-Automaten, der typischerweise zur Implementierung kontextfreier Grammatiken verwendet wird) jetzt nicht nur zum Zählen von Klammern gebraucht, sondern auch zur Gewährleistung ihrer konsistenten Verwendung. Demzufolge leistet der Interaktionsgraph in Abb. 3.31 *nicht* das gewünschte, weil er beispielsweise den inkonsistenten arithmetischen Ausdruck $\langle \mathsf{L}_1, \mathsf{L}_2, \mathsf{NUM}, \mathsf{R}_1, \mathsf{R}_2 \rangle$ akzeptieren würde. Tatsächlich erscheint es schwierig, wenn nicht unmöglich, die konsistente Verwendung der beiden Klammerarten mit Hilfe eines Interaktionsausdrucks zu erzwingen.

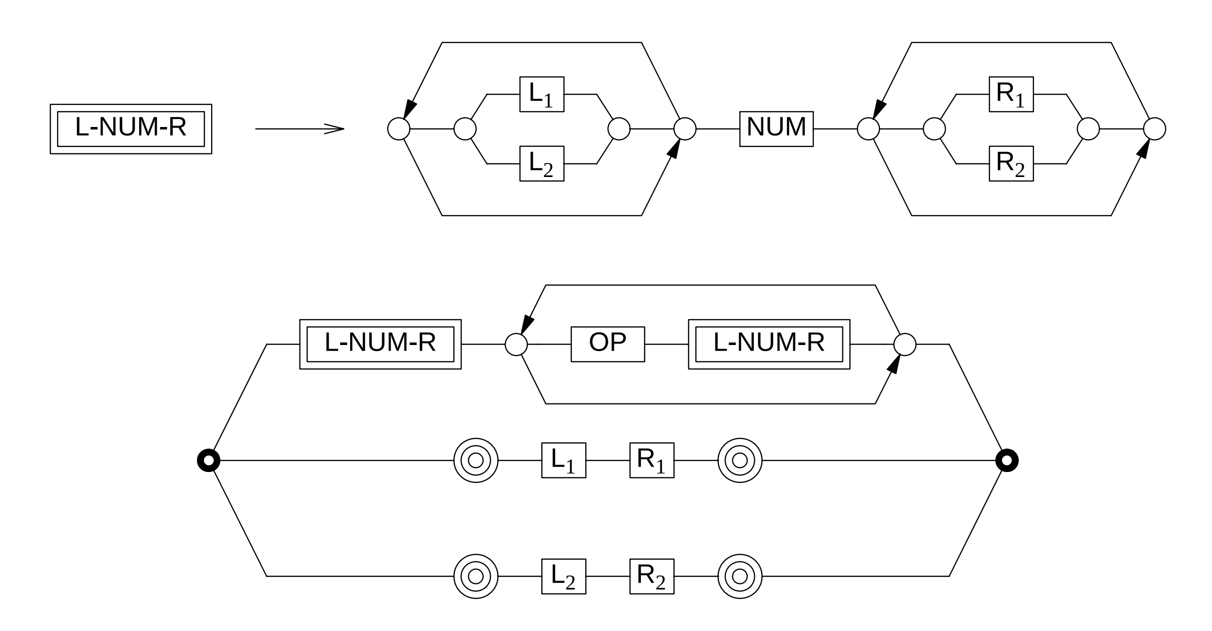


Abbildung 3.31: Fehlerhafter Interaktionsgraph zur Lösung des Klammerproblems

3.5.3.4 Unvergleichbarkeit von Interaktionsausdrücken und kontextfreien Grammatiken?

Betrachtet man die im folgenden Kapitel entwickelte operationale Semantik und Implementierung von Interaktionsausdrücken etwas genauer, so wird dieser Eindruck bestätigt. Die in § 4.5 definierten Zustände[13] besitzen zwar eine *hierarchische* Struktur, ihre *Tiefe* wird jedoch statisch durch die Struktur des vorgegebenen Ausdrucks bestimmt (vgl. auch § 4.7.1.1), d. h. ein „Zustandsbaum“ kann nicht dynamisch in die Tiefe wachsen. Ein dynamisches Wachstum in die *Breite* ist zwar möglich, allerdings werden hierfür immer *Mengen* (oder Multimengen) von Teilzuständen verwendet, bei denen die *Reihenfolge* der Elemente *irrelevant* ist. Somit erscheint es schwierig, wenn nicht unmöglich, mit Hilfe derartiger Zustände das Konzept eines Stacks, d. h. eines *Last-in-first-out-Containers* zu simulieren, das man zur Lösung des Klammerproblems offensichtlich benötigt.

Zwar läßt sich ein Stack auch mit Hilfe von *Zählern* simulieren (man kodiert den Stackinhalt als natürliche Zahl zur Basis k, wenn k die Kardinalität des Stackalphabets bezeichnet [Hopcroft90]), und das Konzept eines Zählers kann prinzipiell mit Hilfe einer parallelen Iteration verwirklicht werden (vgl. § 3.5.3.1 und § 3.5.3.2), indem man die *Kardinalität* eines bestimmten Zustand/Wert-Paars in einer Alternative als Zähler interpretiert (vgl. § 4.5.7.4). Allerdings sind die Möglichkeiten zur *Manipulation* solcher Zähler sehr begrenzt: Bei der Verarbeitung einer Aktion, d. h. bei einem einzelnen Zustandsübergang, werden Kardinalitäten von Zustand/Wert-Paaren lediglich *inkrementiert* oder *dekrementiert*; eine *Multiplikation* oder *Division* eines Zählers mit einem konstanten Faktor (bzw. eine

[13] Zum Verständnis der folgenden Überlegungen ist zumindest die Kenntnis dieses Abschnitts erforderlich.

äquivalente *mehrfache* Addition oder Subtraktion eines Faktors), die man zur Simulation eines Stacks benötigt [Hopcroft90], ist innerhalb *eines* Zustandsübergangs jedoch *nicht* möglich.

Diese Plausibilitätsbetrachtungen stellen natürlich keinen wirklichen Beweis für die Vermutung dar, daß sich das Klammerproblem mit Interaktionsausdrücken nicht lösen läßt und sie somit *teilweise ausdrucksschwächer* als kontextfreie Grammatiken sind. Ein solcher Beweis könnte möglicherweise gelingen, wenn man auch für „Interaktionssprachen" eine Charakterisierung im Sinne eines *Pumping-Lemmas* findet, mit dessen Hilfe dann durch Widerspruch gezeigt werden könnte, daß bestimmte kontextfreie Sprachen keine Interaktionssprachen sind. Da derartige Überlegungen für den praktischen Einsatz von Interaktionsausdrücken jedoch nur eine untergeordnete Rolle spielen, wurden sie im Rahmen dieser Arbeit nicht weiter vertieft.

3.5.3.5 Bezug zu Ereignis- und Flußausdrücken

Es sollte an dieser Stelle jedoch erwähnt werden, daß für einen eng verwandten Ansatz (Flußausdrücke, vgl. § 6.2.3) ursprünglich genau dieselbe Vermutung geäußert wurde [Shaw78], später aber gezeigt werden konnte, daß der Formalismus (ebenso wie Ereignisausdrücke, die im selben Abschnitt beschrieben werden) sogar *berechnungsuniversell* ist, d. h. daß jede Sprache, die von einer *Turingmaschine* erkannt werden kann, auch mit Hilfe eines Ereignis- oder Flußausdrucks beschrieben werden kann [Ogden78, Araki81a]. Zum Beweis wird jeweils gezeigt, daß sich eine *Zweizählermaschine* (von der man weiß, daß sie äquivalent zu einer Turingmaschine ist [Hopcroft90]) stets mit Hilfe eines Ereignis- oder Flußausdrucks simulieren läßt. Allerdings basieren die wesentlichen Teile dieser Beweise auf der Verwendung sogenannter *Synchronisationssymbole*, die es so in Interaktionsausdrücken nicht gibt und die sich *vermutlich* auch nicht mit anderen Konstrukten simulieren lassen.

3.5.4 Zusammenfassung

Zusammenfassend läßt sich festhalten, daß Interaktionsausdrücke *echt ausdrucksstärker* als reguläre Ausdrücke und *teilweise* auch ausdrucksstärker als kontextfreie Grammatiken sind. *Vermutlich* gibt es jedoch auch kontextfreie Sprachen, die sich *nicht* mit Hilfe von Interaktionsausdrücken beschreiben lassen, d. h. vermutlich sind Interaktionsausdrücke und kontextfreie Grammatiken *nicht vergleichbar*.

Anmerkung: Natürlich könnte man die Ausdrucksmächtigkeit von kontextfreien Grammatiken sofort übertreffen, wenn man *rekursive Makrodefinitionen* erlauben würde, da der resultierende Formalismus kontextfreie Grammatiken um die Konzepte *Parallelität* und *Konjunktion* erweitern würde. Abgesehen von den grundsätzlichen Bedenken, daß rekursive Formulierungen für mathematisch ungeübte Anwender nicht ohne weiteres beherrschbar sind (vgl. auch § 2.8.2.5), war es allerdings nie ein Ziel dieser Arbeit, einen derart ausdrucksstarken Formalismus zu entwickeln. Vielmehr wurde versucht, ausgehend von regulären Ausdrücken als Basisformalismus und motiviert durch *konkrete Anwendungsbeispiele*, mit möglichst wenigen zusätzlichen Operatoren und Konzepten einen in sich geschlossenen Formalismus zu entwickeln, mit dem eine Vielzahl praktischer Synchronisationsprobleme zufriedenstellend gelöst werden kann. In diesem Entwicklungsprozeß wurden beispielsweise die parallele Iteration und der Synchronisationsoperator als essentielle Konstrukte identifiziert, während das Konzept der Rekursion zu keinem Zeitpunkt vermißt wurde.

Kapitel 4
Implementierung von Interaktionsausdrücken

4.1 Einleitung

4.1.1 Aufgabenstellung

Nachdem Interaktionsausdrücke und -graphen in den vorangegangenen Kapiteln sowohl anschaulich eingeführt als auch formal definiert worden sind, soll im vorliegenden Kapitel eine konkrete *Implementierung* des Formalismus entwickelt bzw. vorgestellt werden, d. h. ein Programm, das in der Lage ist, für einen beliebigen Ausdruck x die folgenden *Probleme* − möglichst effizient − zu lösen:

Wort- und Teilwortproblem: Für ein Wort $w \in \Sigma^*$ soll entschieden werden, ob es ein vollständiges oder partielles Wort des Ausdrucks x darstellt, d. h. ob $w \in \Phi(x)$ oder $w \in \Psi(x)$ gilt. Aufgrund der Inklusion $\Phi(x) \subseteq \Psi(x)$ (vgl. § 3.4.4) gibt es auf diese Frage genau drei mögliche Antworten:

1. w ist kein partielles und damit auch kein vollständiges Wort von x;
2. w ist ein partielles, aber kein vollständiges Wort von x;
3. w ist ein vollständiges und damit auch ein partielles Wort von x.

Aktionsproblem: Für eine konkrete Aktion $a_1 \in \Sigma$ soll entschieden werden, ob sie im *initialen Zustand* des Ausdrucks x ausgeführt werden darf, d. h. ob das Wort $\langle a_1 \rangle$ ein partielles Wort von x darstellt. Wenn dies der Fall ist, soll für eine zweite Aktion $a_2 \in \Sigma$ entschieden werden, ob sie im Anschluß an a_1 ausgeführt werden darf, d. h. ob das Wort $\langle a_1, a_2 \rangle$ ein partielles Wort von x darstellt, usw. Zur Lösung dieses Problems sollen also *sukzessive* (d. h. typischerweise interaktiv) einzelne Aktionen $a_1, a_2, \ldots$ eingelesen und für jede Aktion a_i ausgegeben werden, ob sie im *aktuellen Zustand der Verarbeitung* zulässig ist, d. h. ob das Wort $\langle a_1, \ldots, a_i \rangle$ ein partielles Wort des Ausdrucks x darstellt.

Bestimmung zulässiger Aktionen: Zusätzlich zum Aktionsproblem soll nach jedem Verarbeitungsschritt die Menge *aller* Aktionen ausgegeben werden, die im *nächsten Schritt* zulässig sind. Nach dem Start des Programms soll also die Menge aller Aktionen a_1 bestimmt werden, die im ersten Schritt ausgeführt werden dürfen. Nach Eingabe und Verarbeitung einer solchen Aktion, soll die Menge aller Aktionen a_2 ausgegeben werden, die im zweiten Schritt zulässig sind, usw.

Offensichtlich läßt sich das Aktionsproblem, wie oben beschrieben, direkt auf das Teilwortproblem zurückführen und umgekehrt. Auch die Bestimmung der im nächsten Schritt zulässigen Aktionen läßt sich *theoretisch* z. B. auf das Aktionsproblem reduzieren, indem man für jede Aktion $a \in \Sigma$ überprüft, ob sie im nächsten Schritt zulässig ist oder nicht. Da die Menge Σ jedoch potentiell unendlich groß ist, ist diese Reduktion *praktisch* nicht anwendbar.

Für praktische Anwendungen von Interaktionsausdrücken bzw. -graphen, wie sie z. B. im nachfolgenden Kapitel 5 beschrieben werden, ist die Lösung des *Wort-* und *Teilwortproblems* von untergeordneter Bedeutung, weil man meist keine *abgeschlossene* Folge von Aktionen vorliegen hat, für die *nachträglich* entschieden werden soll, ob es sich um ein partielles oder vollständiges Wort des Ausdrucks x handelt. Vielmehr werden Aktionen normalerweise *sukzessive* ausgeführt, und es muß jeweils *sofort* entschieden werden, ob eine bestimmte Aktion im aktuellen Zustand der Verarbeitung zulässig ist oder nicht. Aus diesem Grund kommt dem *Aktionsproblem* in praktischen Anwendungen die größte Bedeutung zu.

Die Bestimmung der im nächsten Schritt zulässigen Aktionen mag auf den ersten Blick ebenfalls sehr wichtig erscheinen, erweist sich für viele Anwendungsgebiete jedoch als unnötig. Verwendet

man Interaktionsausdrücke z. B. zur Synchronisation der Prozeduren eines parallelen Programms (vgl. § 5.3), so genügt es völlig, wenn für jede *einzelne*, konkret aufgerufene Prozedur entschieden werden kann, ob ihre Ausführung momentan zulässig ist oder nicht; die Menge *aller* momentan zulässigen Prozeduraufrufe ist ohne Bedeutung.

Auch beim Einsatz von Interaktionsgraphen zur (Spezifikation und) Implementierung von Inter-Workflow-Abhängigkeiten (vgl. § 5.5) genügt es, wenn jeweils für *einzelne* konkrete Aktivitäten entschieden werden kann, ob sie momentan gestartet werden dürfen oder nicht, da die Menge *aller* potentiell zulässigen Aktivitäten von einem Workflow-Management-System bestimmt wird.

Aus diesen Gründen kann die Bestimmung der im nächsten Schritt zulässigen Aktionen als *optionale* „Nice-to-have-Komponente" einer Implementierung von Interaktionsausdrücken betrachtet werden, auf die im weiteren nicht näher eingegangen wird.

Primäres Ziel des Kapitels ist also die Entwicklung einer Software-Komponente, die das Aktionsproblem für einen gegebenen Interaktionsausdruck möglichst effizient lösen kann. Quasi als Abfallprodukt wird hierbei auch eine algorithmische Lösung des Wort- und Teilwortproblems anfallen. Da das vorliegende Kapitel aber noch zum anwendungsunabhängigen Grundlagenteil der Arbeit gehört (Ebene 3 in Abb. 1.8), wird hier noch nicht näher erläutert, wie die entwickelte Software-Komponente tatsächlich zur Koordination oder Synchronisation paralleler Workflows oder ähnlicher Prozesse eingesetzt werden kann. Diese Aspekte werden erst in Kapitel 5 behandelt.

4.1.2 Überblick

Um zu einer praktisch einsetzbaren Implementierung von Interaktionsausdrücken zu gelangen, muß zunächst (§ 4.2) eine „maschinenlesbare" *Syntax* von Interaktionsausdrücken festgelegt werden, die es erlaubt, Ausdrücke in einer *linearen Notation* unter Verwendung eines üblichen *Computerzeichensatzes* darzustellen. Anschließend (§ 4.3) wird die *interne Repräsentation* von Ausdrücken mit Hilfe von *Operatorbäumen* eingeführt.

Nach diesen Vorarbeiten wird in § 4.4 der Versuch unternommen, die in Kapitel 3 definierte *formale Semantik* von Interaktionsausdrücken direkt implementierungstechnisch umzusetzen. Obwohl der hieraus resultierende Algorithmus zweifellos kompakt und elegant ist, besitzt er den alles entscheidenden Nachteil, daß er selbst für sehr einfache Ausdrücke *exponentielle Laufzeit* (bzgl. der Länge der zu verarbeitenden Aktionsfolge) besitzt.

Da dies für realistische Anwendungen kaum akzeptabel ist, wird in § 4.5 eine *operationale Semantik* von Interaktionsausdrücken eingeführt, die auf hierarchisch strukturierten *Zuständen*, zugehörigen *Zustandsprädikaten* sowie *Zustandsübergängen* beruht. Dieses Zustandsmodell, für das in Anhang B gezeigt wird, daß es äquivalent zur formalen Semantik von Interaktionsausdrücken ist, wird in § 4.6 implementierungstechnisch umgesetzt und in § 4.7 bezüglich seiner *Komplexität* untersucht. Hierbei ergibt sich zwar, daß der so entwickelte Algorithmus für gezielt konstruierte *bösartige Ausdrücke* nach wie vor exponentielle Laufzeit besitzt, daß *praktisch relevante Ausdrücke* (wie z. B. sämtliche in dieser Arbeit vorgestellten Beispielausdrücke) aber mit *polynomieller Komplexität* verarbeitet werden können.

Abschnitt 4.8 enthält einen Rückblick auf die Implementierung und faßt die für diesen Erfolg wesentlichen Faktoren noch einmal kurz zusammen.

Abbildung 4.1 skizziert in groben Zügen die Gesamtarchitektur des zu entwickelnden Programms, das aus vier Modulen besteht, die in den angegebenen Abschnitten mehr oder weniger detailliert beschrieben werden. Die von unten nach oben gerichteten gestrichelten Pfeile deuten *Import-Beziehungen* zwischen Modulen an, während die übrigen Pfeile anzeigen, wie ein gegebener Ausdruck nacheinander von den einzelnen Modulen verarbeitet wird und welche *Datenstrukturen* hierbei ausgetauscht werden: Das Modul `parse` verwandelt einen gegebenen Ausdruck (in der Abbildung exemplarisch $(a \circledcirc b) \bigcirc c$), der ihm vom Hauptprogramm übergeben wird, in einen äquivalenten Operatorbaum, der vom Modul `state` sowohl zur Bestimmung des *initialen Zustands* als auch zur Berechnung der *Folgezustände* des Ausdrucks verwendet wird. Durch Auswertung dieser Zustände ist das Hauptpro-

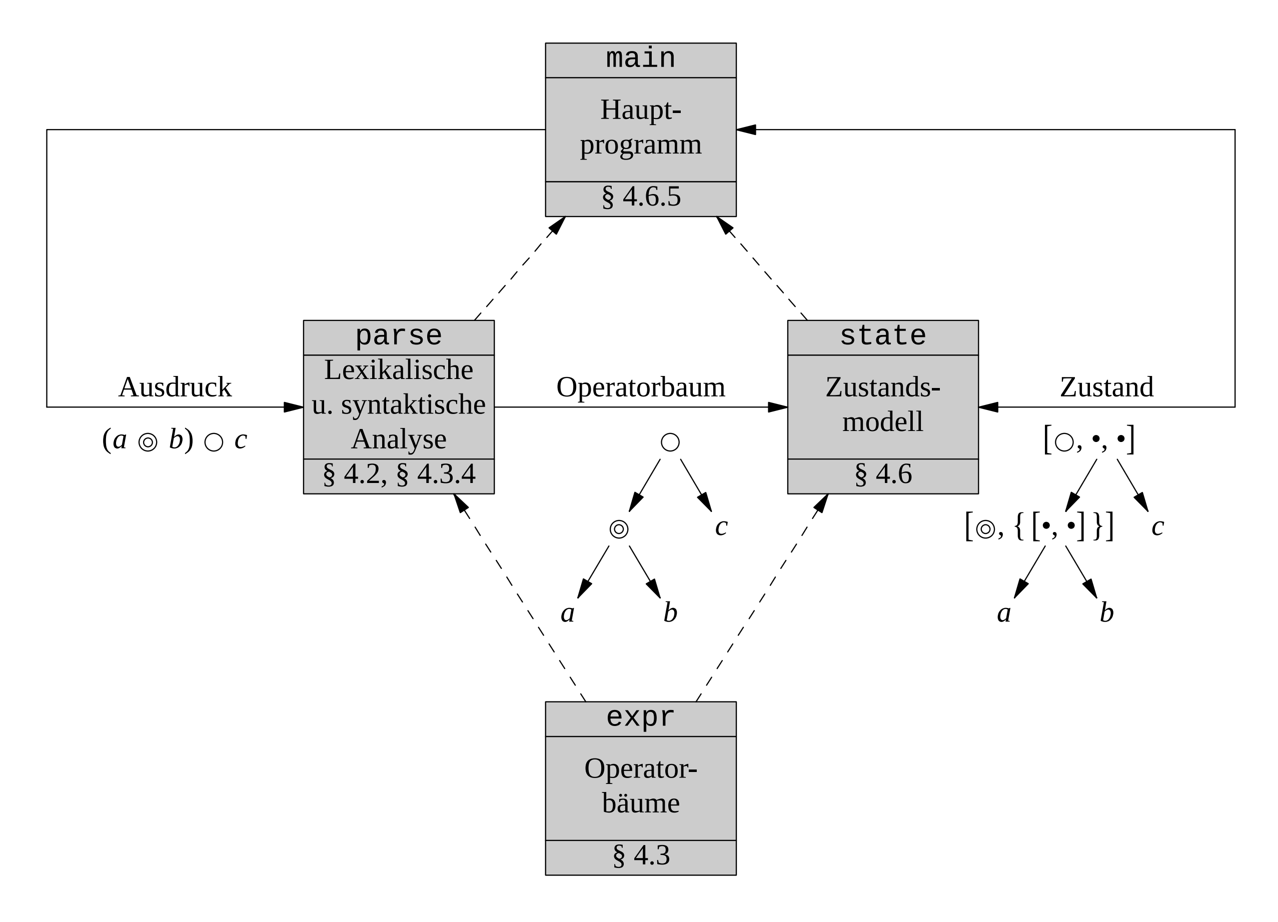


Abbildung 4.1: Modulstruktur

gramm (Modul `main`) in der Lage, das zuvor beschriebene Wort-, Teilwort- und Aktionsproblem zu lösen.

Die Tatsache, daß das vorliegende Kapitel − einschließlich der zugehörigen Anhänge A und B − mit Abstand das längste in dieser Arbeit darstellt, verdeutlicht, daß die Entwicklung einer vollständigen, effizienten und korrekten Implementierung von Interaktionsausdrücken − einschließlich des hierfür erforderlichen theoretischen Unterbaus − einen wesentlichen Teil dieser Arbeit ausmacht.

4.2 Syntax von Interaktionsausdrücken

4.2.1 Aktionen

Die Grundmengen Λ, Π und Ω (vgl. § 3.2.1.1) werden durch die folgenden *lexikalischen Grundelemente* (engl. tokens) repräsentiert:

- Ein *Aktionsname* $a_0 \in \Lambda$ ist entweder ein *Bezeichner* (engl. identifier), wie man ihn aus Programmiersprachen wie z. B. Modula oder C kennt (d. h. eine Folge von Buchstaben und ggf. Ziffern, die mit einem Buchstaben beginnt), oder eine beliebige *Zeichenkette*, die in Anführungszeichen eingeschlossen ist (d. h. eine *Stringkonstante*). Dadurch ist es − ähnlich wie in der Datenbanksprache SQL [Date98] − möglich, Aktionen nicht nur mit einfachen Bezeichnern, wie z. B. `Aufstellen`, `Abbauen` oder `KopieA4`, zu benennen, sondern auch „ausgefallenere“ Namen, wie z. B. `"1DM"` (mit einer Ziffer am Anfang) oder `"Kopie A4"` (mit einer Leerstelle), zu verwenden.

- Ein *Quantorparameter* $p \in \Pi$ entspricht aus zwei Gründen einem einfachen Bezeichner: Zum einen besteht hier normalerweise keine Notwendigkeit, komplexere Namen zu verwenden, zum anderen können Parameter und Werte (siehe unten) auf diese Weise syntaktisch unterschieden werden.
- Ein *konkreter Wert* $\omega \in \Omega$ ist entweder eine Stringkonstante oder eine *natürliche Zahl*, wie z. B. `"Maier"`, `"sono"` oder `4711`.

Mit Hilfe (erweiterter) regulärer Ausdrücke, wie sie beispielsweise vom Scanner-Generator *Lex* [Aho88] unterstützt werden, können die lexikalischen Grundelemente Bezeichner (`Ident`), Zahl (`Number`) und Zeichenkette (`String`) wie folgt definiert werden:[1]

```
Ident   [A-Za-z][A-Za-z0-9]*
Number  [1-9][0-9]*
String  ["][^"]*["]
```

Eine abstrakte Aktion $a \in \Gamma$ ist formal als (n+1)-Tupel $[a_0, a_1, \ldots, a_n]$ mit einem Aktionsnamen $a_0 \in \Lambda$ und Argumenten $a_1, \ldots, a_n \in \Pi \cup \Omega$ definiert (vgl. § 3.2.1.2). Da diese Tupelnotation in praktischen Anwendungen jedoch nicht gebräuchlich ist, wird im folgenden die äquivalente „Funktionsnotation“ $a_0(a_1, \ldots, a_n)$ verwendet, bei der der Aktionsname a_0 außerhalb der Klammern steht. Außerdem wird vereinbart, daß eine leere Argumentliste entfallen soll, d. h. daß die Notation a_0 anstelle von $a_0()$ verwendet wird (vgl. auch § 3.2.1.4). Dies wird durch die in Abb. 4.2 dargestellten Produktionsregeln in BNF-Notation beschrieben.[2]

```
                                /* Eine Aktion besteht aus */
act     : name                  /* einem Namen und */
        | name '(' args ')'     /* einer optionalen Argumentliste. */

                                /* Ein Aktionsname ist */
name    : Ident                 /* entweder ein Bezeichner */
        | String                /* oder eine Zeichenkette. */

                                /* Eine Argumentliste besteht aus */
args    : arg                   /* einem oder */
        | args ',' arg          /* mehreren Argumenten. */

                                /* Ein Argument ist */
arg     : par                   /* entweder ein Parameter */
        | val                   /* oder ein Wert. */

                                /* Ein Quantorparameter ist */
par     : Ident                 /* ein Bezeichner. */

                                /* Ein Wert ist */
val     : Number                /* entweder eine Zahl */
        | String                /* oder eine Zeichenkette. */
```

Abbildung 4.2: Produktionsregeln für Aktionen

[1] Mit eckigen Klammern werden in Lex *Zeichenklassen* beschrieben: Beispielsweise repräsentiert `[A-Za-z]` alle Zeichen von `A` bis `Z` und von `a` bis `z`, d. h. alle Buchstaben. `["]` repräsentiert ein Anführungszeichen; die eckigen Klammern sind hier erforderlich, damit das Anführungszeichen nicht als Lex-Metazeichen interpretiert wird. `[^"]` schließlich steht für die Menge aller Zeichen *außer* dem Anführungszeichen.

[2] Konkret wird die Syntax des Parser-Generators *Yacc* [Aho75, Aho88, Kernighan86] verwendet, bei der Bezeichner entweder für Grammatikbegriffe oder für Terminalsymbole stehen; letztere werden per Konvention groß geschrieben. Zeichen in einfachen Anführungszeichen werden ebenfalls als Terminalsymbole interpretiert.

4.2.2 Ersatzdarstellung und Vorrang von Operatoren

Die formalen Operatorsymbole –, ○, ⊖ usw. werden gemäß Tab. 4.3 durch Zeichen ersetzt, die in jedem gebräuchlichen Computer-Zeichensatz (wie z. B. ASCII) enthalten sind. Außerdem wird vereinbart, daß der *Vorrang* der Operatoren in der Tabelle gruppenweise von oben nach unten *abnimmt*, d. h. daß die unären Operatoren (?, * und #) *stärker binden* als die sequentielle Komposition (-), diese wiederum stärker als die parallele Komposition (+) usw.

Operator	*Zeichen*	*Bedeutung*
⌓	?	Option
⊖	*	Sequentielle Iteration
⊚	#	Parallele Iteration
–	-	Sequentielle Komposition
◎	+	Parallele Komposition
○	|	Disjunktion
●	&	Konjunktion
o	@	Synchronisation

Tabelle 4.3: Ersatzdarstellungen für Operatoren

Bei der Wahl der Ersatzdarstellungen wurde versucht, mit möglichst vielen gebräuchlichen Formalismen, Konventionen und Programmen (wie z. B. den Unix-Werkzeugen *awk*, *grep* und *lex*) kompatibel zu sein und gleichzeitig eine möglichst einprägsame Systematik zu verwenden. Die folgenden Anmerkungen sollen hierbei als Merkhilfe dienen:

Sequentielle Komposition: Das Zeichen „-“ entspricht unmittelbar dem formalen Operator „–“ sowie der graphischen Darstellung von Sequenzen mit Hilfe waagrechter Linien.

Sequentielle Iteration: Das Zeichen * wird sowohl in der Literatur als auch in Programmen, die mit regulären Ausdrücken arbeiten, als Operator für die sequentielle Hülle verwendet. Als weitere Merkhilfe kann dienen, daß das Zeichen * optisch aus mehreren Minuszeichen (sequentielle Komposition) zusammengesetzt ist.

Parallele Komposition: Das Zeichen +, das umgangssprachlich oft als *und* gesprochen wird, soll andeuten, daß bei einer parallelen Komposition $y \circledcirc z$ (im Gegensatz zu einer Disjunktion) die Teilgraphen *y und z* durchlaufen werden müssen. Als weitere Merkhilfe kann dienen, daß das Zeichen + aus *zwei* Strichen zusammengesetzt ist, ebenso wie der Operator ◎ aus *zwei* Kreisen besteht.

Parallele Iteration: In Analogie zur sequentiellen Iteration ist das Zeichen # zur Notation der parallelen Iteration optisch aus mehreren Pluszeichen (parallele Komposition) zusammengesetzt.

Disjunktion: Das Zeichen | wird z. B. bei der (E)BNF-Notation von Grammatiken [Schöning95], den oben genannten Unix-Werkzeugen sowie den Programmiersprachen C und C++ als logischer Oder-Operator verwendet. Als weitere Merkhilfe kann dienen, daß das Zeichen | aus *einem* Strich besteht, ebenso wie der Operator ○ aus *einem* Kreis besteht.

Konjunktion: Das Zeichen & stellt (auch umgangssprachlich) ein allgemein übliches Symbol für einen logischen Und-Operator dar.

Synchronisation: Das Zeichen `@`, üblicherweise als *at* gesprochen, könnte man auch als Abkürzung von *and* auffassen und so zum Ausdruck bringen, daß auch die Synchronisation eine Art logischer Und-Verknüpfung darstellt.

Option: Das Zeichen `?` zeigt an, daß die Ausführung eines Ausdrucks „fraglich", d. h. optional ist. In den oben erwähnten Unix-Werkzeugen wird das Zeichen mit derselben Bedeutung verwendet.

Die Tatsache, daß die Operatoren `|` und `+` in manchen theoretischen Arbeiten gerade mit entgegengesetzter Bedeutung verwendet werden [Milner80, Hennessy88, Baeten90], ist zwar bedauerlich, läßt sich bei der großen Vielfalt verwandter Formalismen aber kaum vermeiden.

Bei der Festlegung des Operatorvorrangs wurde darauf geachtet, daß man bei häufig auftretenden Operatorkombinationen mit möglichst wenig expliziten Klammern auskommt. Aus diesem Grund stehen beispielsweise die Booleschen Operatoren `|`, `&` und `@` *in dieser Reihenfolge* am Ende der Vorrangskala. Der hohe Vorrang der unären Operatoren widerspricht diesem Prinzip in gewisser Weise, da Iterationen so fast immer Klammern erfordern, entspricht aber üblichen Gepflogenheiten. Außerdem ist nicht klar, an welcher Stelle der Skala die Operatoren besser „aufgehoben" wären.

4.2.3 Klammern

Klammern können wie üblich zur expliziten Vorrangregelung oder zur Verbesserung der Lesbarkeit von Ausdrücken eingesetzt werden. Um bei mehrfach verschachtelten Klammerausdrücken die Übersichtlichkeit zu erhöhen, können neben den üblichen *runden* Klammern `(...)` auch *eckige* `[...]` und *geschweifte* Klammern `{...}` zur Gruppierung von Ausdrücken verwendet werden.

4.2.4 Multiplikatoren und Quantoren

„Große Operatoren" mit hoch- und tiefgestellten Termen, wie z. B. $\overset{5}{\circledcirc}$ oder $\underset{p}{\bigcirc}$ zur Darstellung von Multiplikatoren und Quantoren, werden wie folgt linearisiert:

- Ein *tiefgestellter* Term wird, wie ein Feldindex in Programmiersprachen, in *eckige* Klammern eingeschlossen. Demnach ist `|[p]` beispielsweise die Ersatzdarstellung für den Quantor $\underset{p}{\bigcirc}$.
- Ein *hochgestellter* Term wird analog in *geschweifte* Klammern eingeschlossen. Der Multiplikator $\overset{5}{\circledcirc}$ kann daher in der Form `+{5}` notiert werden.

Da Multiplikatoren und Quantoren syntaktisch wie unäre Operatoren verwendet werden, besitzen sie denselben Vorrang wie diese.

Anmerkung: Auf den ersten Blick mag es verwirrend erscheinen, daß eckige und geschweifte Klammern auch zur Gruppierung von Ausdrücken verwendet werden können (vgl. § 4.2.3), ihre Bedeutung also syntaktisch überladen ist. Allerdings sollte man hierbei bedenken, daß auch runde Klammern − wie in den meisten Programmiersprachen − diese Eigenschaft besitzen: zum einen dienen sie zur Kennzeichnung von Parameterlisten (vgl. § 4.2.1), zum anderen zur Gruppierung von Ausdrücken. Dies wird jedoch üblicherweise nicht als Problem empfunden.

Die für die Theorie hilfreiche Konvention, daß Quantorparameter *eindeutig* sein müssen (vgl. § 3.3.3.1), wird aufgehoben und durch geeignete *Sichtbarkeitsregeln* (engl. scope rules) ersetzt, wie man sie von blockorientierten Programmiersprachen kennt. Das bedeutet, daß formal zwischen *Parameterbezeichnern* (die mehrdeutig sein können) und den eigentlichen *Parametern* (die eindeutig sind) unterschieden wird, und daß der Parameter eines inneren Quantors alle gleichnamigen Parameter umgebender Quantoren verbirgt. Somit bezieht sich das Vorkommen eines bestimmten Parameterbezeichners in der Argumentliste einer Aktion immer auf den Parameter des „kleinsten" umgebenden Quantorausdrucks mit diesem Bezeichner.

4.2.5 Grammatik von Interaktionsausdrücken

Nach diesen Vorbereitungen kann die Syntax von Interaktionsausdrücken mit Hilfe der kontextfreien Grammatik in Abb. 4.4 spezifiziert werden, sofern die in § 4.2.2 erwähnten Vorrangregeln mitberücksichtigt werden.[3]

```
                                  /* Ein Interaktionsausdruck ist */
expr    : act                     /* entweder ein atomarer Ausdruck */

        | '?' expr | expr '?'     /* oder eine Option */
        | '*' expr | expr '*'     /* oder eine sequentielle Iteration */
        | '#' expr | expr '#'     /* oder eine parallele Iteration */

        | expr '-' expr           /* oder eine sequentielle Komposition */
        | expr '+' expr           /* oder eine parallele Komposition */

        | expr '|' expr           /* oder eine Disjunktion */
        | expr '&' expr           /* oder eine Konjunktion */
        | expr '@' expr           /* oder eine Synchronisation */

        | mop '{' factor '}' expr /* oder ein Multiplikatorausdruck */
        | qop '[' par ']' expr    /* oder ein Quantorausdruck */

        | '(' expr ')'            /* oder ein geklammerter Ausdruck. */
        | '[' expr ']'
        | '{' expr '}'
factor  : Number                  /* Ein Faktor ist eine Zahl. */

                                  /* Ein Multiplikator-Operator ist */
mop     : '-' | '+'               /* einer der Operatoren - oder +. */

                                  /* Ein Quantor-Operator ist einer */
qop     : '+' | '|' | '&' | '@'   /* der Operatoren +, |, & oder @. */
```

Abbildung 4.4: Grammatik von Interaktionsausdrücken

Aus Kompatibilitätsgründen zu anderen Formalismen und Programmen können die unären Operatoren ?, * und # sowohl *präfix* (wie in Kapitel 3 eingeführt) als auch *postfix* (wie sonst meist üblich) angewandt werden. Auf eine detaillierte Betrachtung von *Multiplikatorausdrücken* wird im weiteren Verlauf des Kapitels aus Platzgründen verzichtet. Ebenso werden *Makros* im folgenden nicht weiter berücksichtigt, da sich die meisten praktisch relevanten Anwendungen des Abstraktionsprinzips mit Hilfe eines vorgeschalteten *Standard-Makroprozessors* abdecken lassen.

Die Grammatikbegriffe `act` und `par` sowie das Terminalsymbol `Number` wurden bereits in § 4.2.1 definiert.

Anmerkung: Durch die Möglichkeit, unäre Operatoren sowohl präfix als auch postfix anzuwenden, entsteht ein weiteres, bisher nicht betrachtetes Vorrangproblem. Beispielsweise könnte der Ausdruck `*x#` einerseits als `(*x)#` und andererseits als `*(x#)` interpretiert werden. Aufgrund der formalen Eigenschaften der Operatoren ?, * und # (vgl. § 3.4.8.2) sind die beiden Varianten jedoch semantisch äquivalent, so daß auf eine explizite Vorrangregelung für derartige Ausdrücke verzichtet werden kann. Kombiniert man postfix angewandte unäre Operatoren jedoch mit Multiplikatoren oder Quantoren,

[3] Der Parser-Generator Yacc bietet hierfür entsprechende Deklarationsmöglichkeiten an, auf die allerdings nicht näher eingegangen werden soll.

wie z. B. im Ausdruck `|[p] a(p) #`, so besitzen die beiden möglichen Interpretationen − einerseits `(|[p] a(p)) #` und andererseits `|[p] (a(p) #)` − tatsächlich unterschiedliche Bedeutungen (vgl. § 2.7.1.3). Um derartige Mehrdeutigkeiten auszuschließen, wird vereinbart, daß postfix angewandte Operatoren − wie in den Programmiersprachen C und C++ − einen höheren Vorrang als präfix angewandte besitzen sollen. Somit ist der Ausdruck `|[p] a(p) #` syntaktisch äquivalent zum Ausdruck `|[p] (a(p) #)`.

4.3 Interne Repräsentation von Ausdrücken

4.3.1 Datenmodell

Abbildung 4.5 zeigt ein *semantisches Datenmodell* (im Sinne eines *Schemas*) zur Beschreibung von Interaktionsausdrücken. Das Modell besteht aus *Typen*, wie z. B. `Expr` oder `Action`, die durch *Attribute* (oder Beziehungen) miteinander verbunden sind. Ein *einfacher Pfeil* von einem Typ *A* zu einem Typ *B* (wie z. B. von `Expr` nach `Category`) repräsentiert ein *einwertiges* Attribut (d. h. eine 1:1- oder *n*:1-Beziehung), während ein *Doppelpfeil* (wie z. B. von `Action` nach `Param`) ein *mehrwertiges* Attribut (d. h. eine 1:*n*- oder *m*:*n*-Beziehung) des Typs *A* bezeichnet. Attribute sind grundsätzlich *optional*, d. h. bei konkreten Ausprägungen eines bestimmten Typs können einzelne Attribute dieses Typs fehlen. So ist es beispielsweise zulässig, daß bei Objekten vom Typ `Expr` das Attribut `body` fehlt, oder daß Objekte vom Typ `Action` keine Parameter (Attribut `params`) besitzen.[4]

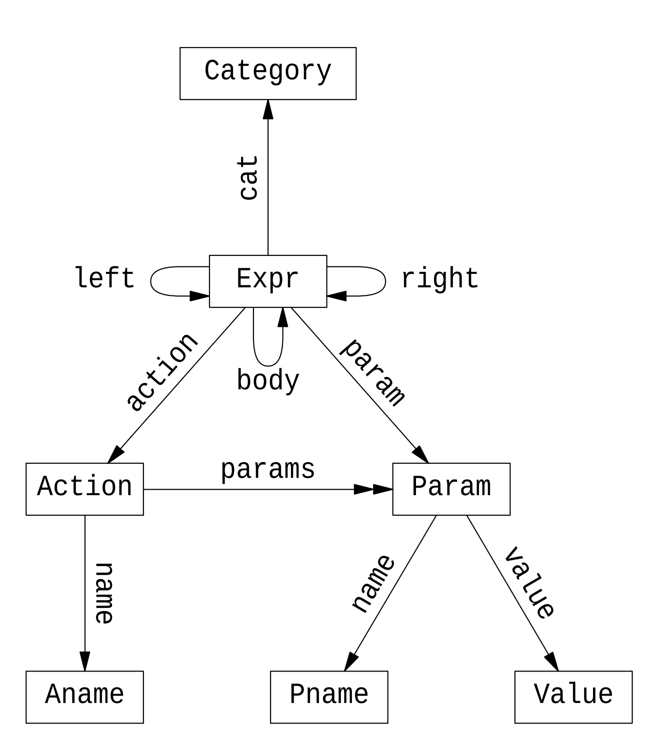


Abbildung 4.5: Semantisches Datenmodell für Interaktionsausdrücke

Ein Typ, der Attribute besitzt, d. h. von dem ein oder mehrere Pfeile ausgehen, wird im folgenden *strukturiert* genannt. In Abb. 4.5 sind dies die Typen `Expr`, `Action` und `Param`. Die übrigen Typen werden als *skalar* bezeichnet.

[4] In der Terminologie relationaler Datenbanksysteme bedeutet das, daß Attribute *Nullwerte* besitzen dürfen [Date98].

Die skalaren Typen `Aname` (Aktionsnamen), `Pname` (Parameternamen) und `Value` (Parameterwerte), die den Mengen Λ, Π und Ω des formalen Modells entsprechen, sind lediglich Synonyme eines vordefinierten Typs `String`, der die Menge aller Zeichenketten (Strings) repräsentiert.

Der Typ `Param` beschreibt einen Parameter bzw. ein Argument einer Aktion. Je nachdem, ob ein Objekt dieses Typs das Attribut `name` oder `value` besitzt, repräsentiert es entweder einen Quantorparameter $p \in \Pi$ oder einen konkreten Wert $\omega \in \Omega$. Sind beide Attribute vorhanden, so wird ein Parameter $p \in \Pi$ beschrieben, dem momentan ein konkreter Wert $\omega \in \Omega$ zugeordnet ist (vgl. § 4.6.3.1).

Der Typ `Action` beschreibt abstrakte Aktionen $a \in \Gamma$, die aus einem Namen vom Typ `Aname` (einwertiges Attribut `name`) und einer Folge von Parametern oder Argumenten vom Typ `Param` (mehrwertiges Attribut `params`) bestehen.

Der Typ `Category` repräsentiert einen *Aufzählungstyp* mit den Elementen bzw. Werten

```
Atom, Sequ, Par, Disj, Conj, Sync, Iter
```

zur Beschreibung von Ausdrucks-Kategorien, deren Verwendung im folgenden näher erläutert wird.

Der Typ `Expr` schließlich dient zur Repräsentation beliebiger Interaktionsausdrücke mit Ausnahme paralleler Iterationen, die gemäß § 3.4.11 durch parallele Quantorausdrücke ersetzt werden können. (Diese Ersetzung wird bereits bei der syntaktischen Analyse eines Ausdrucks vorgenommen.)

Tabelle 4.6 zeigt die Verwendung und Belegung der Attribute `cat`, `action`, `left`, `right`, `body` und `param` des Typs `Expr` für die verschiedenen Ausdrucks-Kategorien. Wie man sieht, verweisen die Attribute `left` und `right` bei einem binären Ausdruck y ◎ z (mit ◎ $\in$ { −, ○, ◎, ⭘, ● }) jeweils „rekursiv" auf die Teilausdrücke y und z (d. h. auf Objekte vom Typ `Expr`, die diese Teilausdrücke repräsentieren), während das Attribut `body` auf den Rumpf y einer sequentiellen Iteration ⊖ y

Bezeichnung	Ausdruck	Attribute					
		`cat`	`action`	`left`	`right`	`body`	`param`
Atomarer Ausdruck	a	`Atom`	a	–	–	–	–
Sequentielle Komp.	$y - z$	`Sequ`	–	y	z	–	–
Parallele Komp.	y ◎ z	`Par`	–	y	z	–	–
Disjunktion	y ○ z	`Disj`	–	y	z	–	–
Konjunktion	y ● z	`Conj`	–	y	z	–	–
Synchronisation	y ⭘ z	`Sync`	–	y	z	–	–
Option	◡ y	`Disj`	–	y	–	–	–
Sequentielle Iteration	⊖ y	`Iter`	–	–	–	y	–
Parallele Iteration	⊚ y	Implementierung als ◎$_p$ ◡ y mit einem anonymen Parameter p					
Disjunktions-Quantorausdruck	○$_p$ y	`Disj`	–	–	–	y	p
Paralleler Quantorausdruck	◎$_p$ y	`Par`	–	–	–	y	p
Konjunktions-Quantorausdruck	●$_p$ y	`Conj`	–	–	–	y	p
Synchronisations-Quantorausdruck	⭘$_p$ y	`Sync`	–	–	–	y	p

Tabelle 4.6: Verwendung von Attributen des Typs `Expr`

oder eines Quantorausdrucks $\underset{p}{\bigcirc}\, y$ (mit $\bigcirc \in \{ \circ, \circledcirc, \boldsymbol{\circ}, \bullet \}$) verweist.[5] Im Fall eines Quantorausdrucks referenziert das Attribut `param` außerdem den Quantorparameter p, d. h. ein entsprechendes Objekt vom Typ `Param`. Eine Option $\circlearrowleft y$ wird als spezielle Disjunktion repräsentiert, bei der der rechte Teilausdruck `right` fehlt (vgl. auch § 4.5.4.3).

4.3.2 Beispiel

Abbildung 4.7 zeigt die interne Repräsentation des Quantorausdrucks

$$\underset{p}{\circledcirc} \circlearrowright \underset{u}{\bigcirc} (\text{abrufen}(p, u) - \text{untersuchen}(p, u))$$

mit der linearen Notation

```
+[p] * |[u] (abrufen(p, u) - untersuchen(p, u)).
```

Im Gegensatz zu Abb. 4.5, repräsentiert ein Rechteck hier keinen Typ, sondern ein *Objekt*, d. h. eine Ausprägung eines strukturierten Typs, während eine Ellipse eine Ausprägung eines skalaren Typs, d. h. einen *skalaren Wert* oder eine *Konstante* darstellt.

Pfeile zwischen Objekten (bzw. zwischen einem Objekt und einer Konstanten) stellen Ausprägungen des Attributs dar, mit dem sie beschriftet sind. Für mehrwertige Attribute, wie z. B. `params`, können von einem Objekt mehrere gleichartige Pfeile ausgehen, deren Beschriftung (wie z. B. `params[1]` und `params[2]`) bei Bedarf eine *Reihenfolge* der Attributwerte eines Objekts bestimmt.

Bei der Repräsentation eines Quantorausdrucks $x \equiv \underset{q}{\bigcirc}\, y$ wird darauf geachtet, daß es *genau ein* Objekt vom Typ `Param` gibt, das den Quantorparameter q repräsentiert. Dieses Objekt wird sowohl vom `Expr`-Objekt, das dem Quantorausdruck x entspricht, als auch von allen `Action`-Objekten des Quantorrumpfs y, die q als Parameter besitzen, als *gemeinsames Objekt* (engl. shared object) referenziert. Beispielsweise wird das `Param`-Objekt mit dem Namen `"u"` sowohl vom `Expr`-Objekt mit der Kategorie `Disj` (das den Disjunktions-Quantor $\underset{u}{\bigcirc}$ repräsentiert) als auch von den beiden `Action`-Objekten (die die Aktionen abrufen(p, u) und untersuchen(p, u) des zugehörigen Quantorrumpfs repräsentieren) referenziert. Auf diese Weise sind Änderungen, die während der Verarbeitung des Ausdrucks an einem Quantorparameter-Objekt vorgenommen werden, sofort für alle Aktions-Objekte sichtbar, die diesen Parameter referenzieren. Dies wird später zur effizienten Konkretisierung von Ausdrücken und Zuständen verwendet (vgl. § 4.6.3.1).

4.3.3 Implementierung des Datenmodells

Der Programmcode in Abb. 4.8, der im folgenden näher erläutert wird, stellt eine direkte implementierungstechnische Umsetzung des Datenmodells aus § 4.3.1 mit Hilfe der Programmiersprache `CH`[6] dar.

4.3.3.1 Offene Typen

Typen und Attribute werden mit den Schlüsselwörtern `type` bzw. `attr` deklariert, während Ausprägungen (engl. instances) eines Aufzählungstyps mit Hilfe des Schlüsselworts `inst` vereinbart werden. Bemerkenswert ist, daß sowohl `attr`- als auch `inst`-Deklarationen für einen Typ *inkrementell* erfolgen können. Das bedeutet, daß ein einmal vereinbarter Typ *nachträglich* (d. h. an einer beliebigen an-

---

[5] Um Speicherplatz zu sparen, könnte man versucht sein, auf das Attribut `body` zu verzichten und den Rumpf eines unären Operators stattdessen im Attribut `left` oder `right` abzulegen. Noch weitergehende Optimierungen würden eventuell auch noch `action` und `param` mit `left` und `right` verschmelzen, da sich implementierungstechnisch hinter allen Attributen lediglich Zeigerwerte verbergen. Derartige Optimierungen, die man etwas eleganter auch mit Hilfe varianter Strukturen modellieren könnte, werden in der zur Implementierung des Datenmodells verwendeten Programmiersprache jedoch automatisch durch das Laufzeitsystem vorgenommen, das dafür sorgt, daß nichtbelegte Attribute eines Objekts auch keinen Speicherplatz verbrauchen (vgl. § A.2.2). Aus diesem Grund kann man bei der Modellierung und Implementierung von Datenstrukturen durchaus „verschwenderisch" mit Attributen umgehen und „jedem Kind seinen eigenen Namen geben".

[6] `CH` ist eine Variante der bekannten Programmiersprache C++, deren wesentliche Konzepte im weiteren Verlauf des Kapitels en passant erläutert werden. Anhang A enthält außerdem eine zusammenfassende Übersicht.

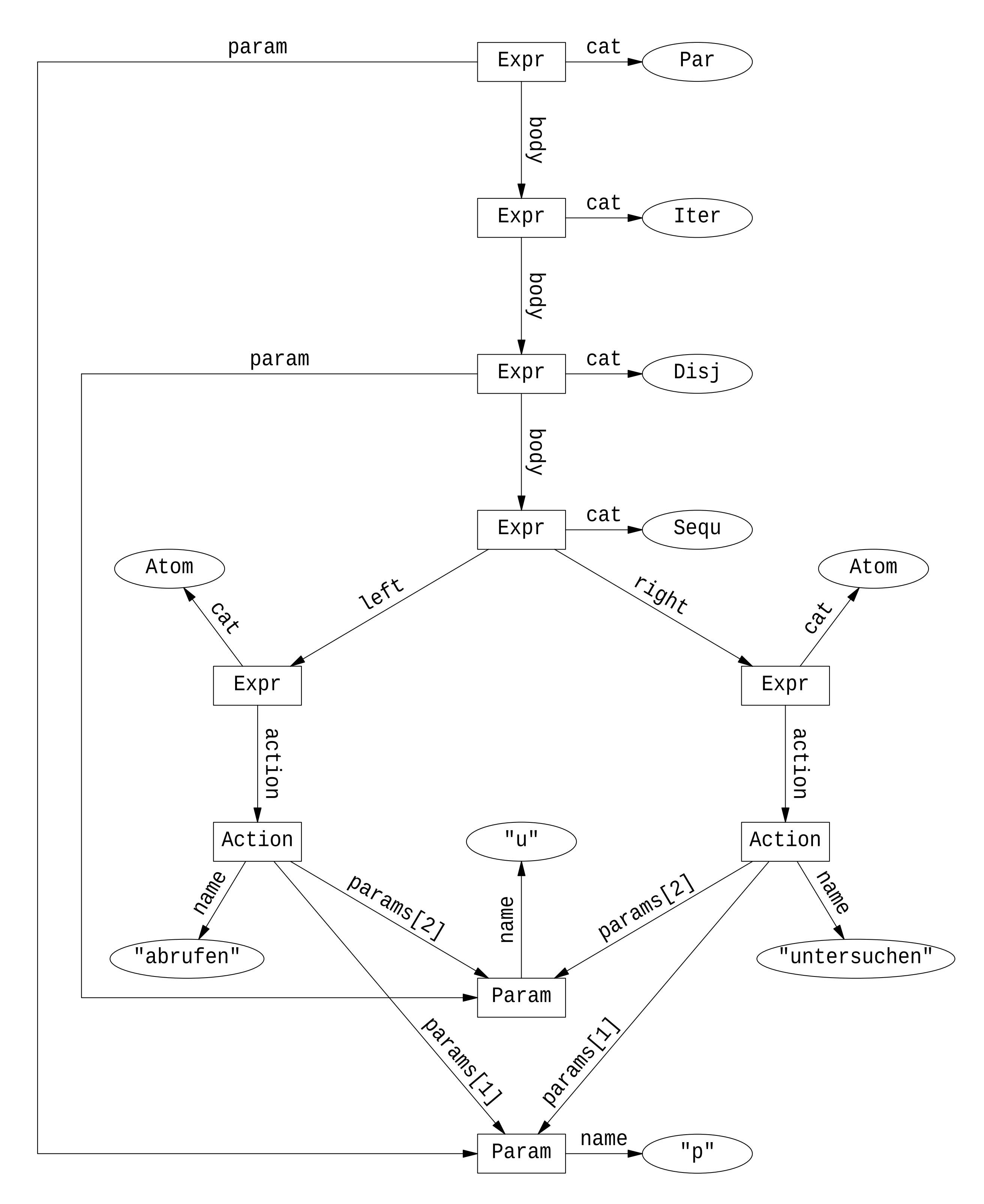


Abbildung 4.7: Interne Repräsentation des Ausdrucks
`+[p] * |[u] (abrufen(p, u) − untersuchen(p, u))`

deren Stelle des Programms) um weitere Attribute oder Konstanten *erweitert* werden kann, ohne daß hierfür − wie bei objektorientierten Programmiersprachen − ein *neuer*, abgeleiteter Typ definiert werden muß. Derartige Typen werden in `CH` als *offene Typen* bezeichnet (vgl. auch § A.1.2 und § A.2.2).

```
// Aktionsnamen, Parameternamen und Parameterwerte.
type Aname = String;
type Pname = String;
type Value = String;

// Parameter.
type Param;
attr name: Param -> Pname;      // Name.
attr value: Param -> Value;     // Wert.

// Aktionen.
type Action;
attr name: Action -> Aname;     // Name.
attr params: Action ->> Param;  // Parameterliste.

// Kategorien.
type Category;
inst Category { Atom, Sequ, Par, Disj, Conj, Sync, Iter };

// Ausdrücke.
type Expr;
attr cat: Expr -> Category;     // Kategorie.
attr action: Expr -> Action;    // Aktion eines atomaren Ausdrucks.
attr left: Expr -> Expr;        // Linker bzw. rechter Operand
attr right: Expr -> Expr;       // eines binären Operators.
attr body: Expr -> Expr;        // Rumpf eines unären Operators.
attr param: Expr -> Param;      // Parameter eines Quantorausdrucks.
```

Abbildung 4.8: Implementierungstechnische Umsetzung des Datenmodells

Eine Attributdeklaration ist daher mit einer Alter-Table-Anweisung in SQL [Date98] vergleichbar, die eine bestehende Tabelle (d. h. in diesem Zusammenhang einen Typ) um ein neues Attribut erweitert.

4.3.3.2 Ein- und mehrwertige Attribute

Die Pfeil- bzw. Doppelpfeil-Notation zur Unterscheidung ein- bzw. mehrwertiger Attribute kann unmittelbar aus der graphischen Darstellung des Datenmodells übernommen werden. Ein *einwertiges* Attribut $a: D \to R$ (wie z. B. `name: Action -> Aname`) entspricht mathematisch einer Funktion a mit Definitionsmenge (engl. domain) D und Wertebereich (engl. range) R. Da Attribute jedoch optional sind, ist diese Funktion u. U. nur für eine Teilmenge $D' \subseteq D$ definiert. Um zu einer vollständig definierten Funktion zu gelangen, wird für Objekte $x \in D \setminus D'$ ein *Defaultwert* $a(x) := \delta(R) \in R$ vereinbart, der abhängig vom konkreten Wertebereich R ist.[7] Für numerische Wertebereiche gilt beispielsweise $\delta(R) = 0$, während für strukturierte Typen $\delta(R) =$ `nil` vereinbart wird, wobei `nil` ein *generisches Objekt* darstellt, das (ähnlich wie die Zeigerkonstante `NIL` in Modula) mit allen strukturierten Typen kompatibel ist, aber keinerlei Attributausprägungen besitzt.[8]

Ein *mehrwertiges* Attribut $a: D \twoheadrightarrow R$ (wie z. B. `params: Action ->> Param`) entspricht mathematisch einer Funktion $a: D \to R^*$, wenn man mit R^* wie gewohnt die Menge aller Folgen $\langle r_1, \ldots, r_n \rangle$ mit $n \in \mathbb{N}_0$ und $r_1, \ldots, r_n \in R$ bezeichnet. Da die leere Folge $\langle\rangle$ ebenfalls zur Menge R^* gehört, müssen fehlende Attributwerte hier nicht gesondert betrachtet werden.

[7] Diese Eigenschaft unterscheidet einen Defaultwert $\delta(R)$ von dem Typ-unabhängigen Nullwert `NULL` in relationalen Datenbanksystemen.
[8] Es gilt also $a($`nil`$) = \delta(R)$ für alle Attribute $a: D \to R$ und insbesondere $a($`nil`$) =$ `nil`, wenn der Wertebereich des Attributs wiederum ein strukturierter Typ ist.

4.3.3.3 Konstruktorfunktionen

Die Funktionen in Abb. 4.9, die jeweils ein geeignet initialisiertes Objekt des Typs `Expr` zurückliefern, implementieren im Prinzip die Tabelle 4.6. Durch den Aufruf der vordefinierten generischen Konstruktorfunktion `Expr()`, die (wie in C++) denselben Namen wie der Typ `Expr` besitzt, wird jeweils ein Objekt dieses Typs erzeugt und mit den Attributausprägungen, die als Argumente übergeben werden, initialisiert. So wird in der Funktion `atom()` beispielsweise ein `Expr`-Objekt erzeugt, dessen Attribute `cat` und `action` mit der Konstanten `Atom` (vom Typ `Category`) bzw. dem Objekt `a` (vom Typ `Action`) initialisiert werden.

```
// Konstruktorfunktion für atomare Ausdrücke a.
Expr atom(Action a) {
    return Expr(cat = Atom, action = a);
}
// Konstruktorfunktion für binäre Ausdrücke y ˜ z.
Expr bin(Category c, Expr y, Expr z) {
    return Expr(cat = c, left = y, right = z);
}
// Konstruktorfunktion für Iterationen *y.
Expr iter(Expr y) {
    return Expr(cat = Iter, body = y);
}
// Konstruktorfunktion für Quantorausdrücke ˜[p] y.
Expr quant(Category c, Param p, Expr y) {
    return Expr(cat = c, param = p, body = y);
}
```

Abbildung 4.9: Konstruktorfunktionen für Ausdrücke

4.3.3.4 Beispiel

Der Programmcode in Abb. 4.10 zeigt die Verwendung dieser Konstruktorfunktionen, um den in Abb. 4.7 dargestellten Operatorbaum aufzubauen. Ebenso wie in C++, können Variablen (wie z. B. `p` oder `call`) bei ihrer Deklaration unmittelbar durch einen Konstruktoraufruf initialisiert werden. Einem mehrwertigen Attribut (wie z. B. `params`) können mit Hilfe des Operators `+=` sukzessive mehrere Attributwerte (im Beispiel `p` und `u`) zugeordnet werden.

4.3.4 Parser für Interaktionsausdrücke

Nachdem nun sowohl die syntaktische Struktur als auch die interne Repräsentation von Interaktionsausdrücken festgelegt ist, kann mit Hilfe von Standard-Compilerbau-Techniken und -Werkzeugen (wie z. B. Lex und Yacc) ohne große Mühe ein Parser für Interaktionsausdrücke konstruiert werden, der einen als Zeichenkette vorliegenden Ausdruck in einen äquivalenten *Operatorbaum* transformiert, der als Grundlage der weiteren Verarbeitung dient.

Wie bereits in Abschnitt 4.3.2 erwähnt wurde, muß bei der Repräsentation von Quantorausdrücken darauf geachtet werden, daß das `Param`-Objekt, das den Quantorparameter repräsentiert, von sämtlichen `Action`-Objekten des Quantorrumpfs (die diesen Parameter enthalten) als *gemeinsames Objekt* referenziert wird.

```
Expr example() {
    // Quantorparameter p und u.
    Param p(name = "p");
    Param u(name = "u");

    // Aktionen "abrufen" und "untersuchen".
    Action call(name = "abrufen", params += p, params += u);
    Action exam(name = "untersuchen", params += p, params += u);

    // Sequenz call − exam.
    Expr s = bin(Sequ, atom(call), atom(exam));

    // Kompletter Ausdruck.
    return quant(Par, p, iter(quant(Disj, u, s)));
}
```

Abbildung 4.10: Verwendung von Konstruktorfunktionen (vgl. Abb. 4.7)

4.4 Implementierung der formalen Semantik

4.4.1 Algorithmus

Der in den Abbildungen 4.11 bis 4.13 dargestellte Algorithmus, dessen prinzipielle Idee aus [Shaw80a] übernommen wurde, stellt den Versuch dar, die durch formale Sprachen definierte Semantik von Interaktionsausdrücken (vgl. § 3.3.4) unmittelbar implementierungstechnisch umzusetzen. In

```
// Wort als Sequenz von Aktionen.
type Word = Seq(Action);

// Überprüfe, ob w ein vollst. Wort des atomaren Ausdrucks x darstellt.
part bool test(Expr x, Word w) if (x/cat == Atom) {
    // w ist genau dann ein vollst. Wort des atomaren Ausdrucks x = a,
    // wenn w == <a> gilt.
    return *w == 1 && w[1] == x/action;
}

// Überprüfe, ob w ein vollst. Wort der Disjunktion x darstellt.
part bool test(Expr x, Word w) if (x/cat == Disj) {
    // w ist genau dann ein vollst. Wort der Disjunktion x = y | z,
    // wenn es ein vollst. Wort von y oder ein vollst. Wort von z ist.
    return test(x/left, w) || test(x/right, w);
}

// Überprüfe, ob w ein vollst. Wort der Konjunktion x darstellt.
part bool test(Expr x, Word w) if (x/cat == Conj) {
    // w ist genau dann ein vollst. Wort der Konjunktion x = y & z,
    // wenn es ein vollst. Wort von y und ein vollst. Wort von z ist.
    return test(x/left, w) && test(x/right, w);
}
```

Abbildung 4.11: Implementierung der formalen Semantik
(Teil 1: Atomare und Boolesche Ausdrücke)

```
// Überprüfe, ob w ein vollst. Wort der sequ. Komp. y − z darstellt.
bool testseq(Expr y, Expr z, Word w, int start) {
    // Zerschneide w an einer beliebigen Stelle i in Teilworte u und v.
    // w ist genau dann ein vollst. Wort der sequ. Komposition y − z,
    // wenn es ein i gibt, für das u ein vollst. Wort von y
    // und v ein vollst. Wort von z ist.
    for (int i = start; i <= *w; i++) {
        Word u = w(1, i);
        Word v = w(i+1, *w);
        if (test(y, u) && test(z, v)) return true;
    }
    return false;
}

// Überprüfe, ob w ein vollst. Wort der sequ. Komposition x darstellt.
part bool test(Expr x, Word w) if (x/cat == Sequ) {
    return testseq(x/left, x/right, w, 0);
}

// Überprüfe, ob w ein vollst. Wort der sequ. Iteration x darstellt.
part bool test(Expr x, Word w) if (x/cat == Iter) {
    // w ist genau dann ein vollst. Wort der sequ. Iteration x = * y,
    // wenn w das leere Wort ist
    // oder wenn w ein vollst. Wort der sequ. Komposition y − x ist.
    return *w == 0 || testseq(x/body, x, w, 1);
}
```

Abbildung 4.12: Implementierung der formalen Semantik
(Teil 2: Sequentielle Komposition und Iteration)

der vorgestellten Form löst er prinzipiell das Wortproblem für elementare Ausdrücke mit Ausnahme der Synchronisation; eine Erweiterung auf beliebige Interaktionsausdrücke sowie zur Lösung des Teilwortproblems wäre jedoch mit moderatem Aufwand möglich.

Der Algorithmus an sich sollte mit Hilfe der eingefügten Kommentare selbsterklärend sein, zum Verständnis einiger syntaktischer Konstrukte sind jedoch die nachfolgenden Erläuterungen erforderlich.

4.4.2 Erläuterungen

4.4.2.1 Dynamische Sequenzen

Die Sprache `CH` (bzw. die zugehörige Standardbibliothek) unterstützt einen generischen Typ `Seq(T)` zur Verwaltung *dynamischer Sequenzen* eines beliebigen Elementtyps `T`. So wird in Abb. 4.11 beispielsweise der Typ `Word` als Synonym für `Seq(Action)`, d. h. als Sequenz von Aktionen vereinbart. Typische Operationen auf derartigen Sequenzen sind (vgl. auch § A.3.2.1):

- der Zugriff auf ein bestimmtes Element mittels Index, wie z. B. `w[1]`;
- die Bestimmung der *Kardinalität* oder Länge einer Sequenz mit Hilfe des Präfixoperators `*`, z. B. `*w`;
- die Extraktion einer *Teilfolge*, wie z. B. `w(1, i)`, die sich vom ersten bis zum `i`-ten Element von `w` erstreckt.

```
// Überpruefe, ob w ein vollst. Wort der par. Komp. y + z darstellt.
bool testpar(Expr y, Expr z, Word w, int start) {
    // Verteile die Zeichen von w anhand des Bitmusters von i
    // auf die Teilworte u und v.
    // w ist genau dann ein vollst. Wort der par. Komposition y + z,
    // wenn es ein i gibt, für das u ein vollst. Wort von y
    // und v ein vollst. Wort von z ist.
    for (int i = start; i <= (1<<*w) − 1; i++) {
        Word u = select(w, i, 1);
        Word v = select(w, i, 0);
        if (test(y, u) && test(z, v)) return true;
    }
    return false;
}

// Überprüfe, ob w ein vollst. Wort der par. Komposition x darstellt.
part bool test(Expr x, Word w) if (x/cat == Par) {
    return testpar(x/left, x/right, w, 0);
}

// Überprüfe, ob w ein vollst. Wort der par. Iteration x darstellt.
part bool test(Expr x, Word w) if (x/cat == ParIter) {
    // w ist genau dann ein vollst. Wort der par. Iteration x = # y,
    // wenn w das leere Wort ist
    // oder wenn w ein vollst. Wort der par. Komposition y + x ist.
    return *w == 0 || testpar(x/body, x, w, 1);
}
```

Abbildung 4.13: Implementierung der formalen Semantik
(Teil 3: Parallele Komposition und Iteration)

4.4.2.2 Zugriff auf Attribute

Der lesende Zugriff auf die Attributwerte eines Objekts erfolgt in `CH` mit Hilfe des *Schrägstrich-Operators* (wie z. B. `x/cat` oder `x/action`), der in etwa dem Punkt- oder Pfeiloperator `->` in C bzw. C++ entspricht.

4.4.2.3 Vergleichsoperatoren

Die Standard-Vergleichsoperatoren (`==`, `!=`, `<`, `<=`, `>`, `>=`) können wie in C++ für benutzerdefinierte Typen *überladen*, d. h. mit einer anwendungsspezifischen Semantik belegt werden. Beispielsweise sollen zwei Aktionen vom Typ `Action` genau dann gleich sein, wenn sie denselben Namen und dieselben Parameter besitzen. Die `CH`-Bibliothek bietet spezielle Mechanismen an, die es erlauben, durch die Definition einer einzigen Vergleichsprozedur `cmp()` alle sechs Vergleichsoperatoren auf einmal für einen bestimmten Typ (wie z. B. `Action`) zu implementieren. Außerdem ist die Prozedur `cmp()` für alle eingebauten Typen der Sprache einschließlich Sequenzen vordefiniert, was die Formulierung neuer Vergleichsvorschriften erheblich vereinfacht. (Vgl. auch § A.2.2 und § A.3.2.5.)

4.4.2.4 Partielle Funktionen

Das Schlüsselwort `part` am Anfang einer Funktionsdefinition leitet eine *partielle* oder *stückweise definierte* Funktion ein, deren Rumpf nur ausgeführt werden darf, wenn die im Funktionskopf nach dem Schlüsselwort `if` formulierte *Bedingung* von den aktuellen Aufrufparametern erfüllt wird. Dies ent-

spricht der in mathematischen Texten gebräuchlichen bedingten Formulierung

$$f(x_1, \ldots, x_n) = g_1(x_1, \ldots, x_n), \quad \text{falls } P_1(x_1, \ldots, x_n),$$

die besagt, daß die Definition $f(x_1, \ldots, x_n) = g_1(x_1, \ldots, x_n)$ nur zutreffend ist, wenn die Funktionsargumente $x_1, \ldots, x_n$ das Prädikat $P_1(x_1, \ldots, x_n)$ erfüllen. Um den Definitionsbereich einer so vereinbarten Funktion f sukzessive auszudehnen, können nach Belieben weitere „Zweige“ mit geeigneten Definitionen $g_2, g_3, \ldots$ und zugehörigen Prädikaten $P_2, P_3, \ldots$ hinzugefügt werden:

$$\begin{array}{ll} f(x_1, \ldots, x_n) = g_2(x_1, \ldots, x_n), & \text{falls } P_2(x_1, \ldots, x_n), \\ f(x_1, \ldots, x_n) = g_3(x_1, \ldots, x_n), & \text{falls } P_3(x_1, \ldots, x_n), \\ \vdots & \vdots \end{array}$$

Zusammengefaßt entspricht dies der Definition

$$f(x_1, \ldots, x_n) = \begin{cases} g_1(x_1, \ldots, x_n), & \text{falls } P_1(x_1, \ldots, x_n), \\ g_2(x_1, \ldots, x_n), & \text{falls } P_2(x_1, \ldots, x_n), \\ \vdots & \vdots \\ h(x_1, \ldots, x_n) & \text{sonst,} \end{cases}$$

bei der noch ein „Else-Zweig“ $h(x_1, \ldots, x_n)$ hinzugefügt wurde, der gewählt werden soll, wenn keines der Prädikate P_i erfüllt ist.

In derselben Art und Weise kann man sich vorstellen, daß alle stückweisen Definitionen der Funktion `test()` in den Abbildungen 4.11 bis 4.13 vom CH-(Prä-)Compiler zu einer virtuellen Gesamtfunktion

```
bool test(Expr x, Word w) {
    if (x/cat == Atom) { ... }
    else if (x/cat == Disj) { ... }
    else if (x/cat == Conj) { ... }
    else if ...
}
```

zusammengefügt werden. Allerdings stellt diese Sicht der Dinge nur ein stark vereinfachtes mentales Modell dar. Die tatsächliche implementierungstechnische Umsetzung, die in § A.3.2.6 skizziert wird, ist eher mit den aus objektorientierten Sprachen bekannten Konzepten *late binding* oder *dynamic dispatching* vergleichbar. Allerdings basiert der Auswahlmechanismus für partielle Funktionen nicht nur (wie bei den meisten derartigen Sprachen) auf dem dynamischen *Typ* eines **einzigen** Objekts, sondern erlaubt vielmehr beliebige *Prädikate* (Boolesche Ausdrücke) über **allen** Argumenten der Funktion. In objektorientierter Terminologie sind partielle Funktionen daher am ehesten mit *Multimethoden* vergleichbar (wie man sie beispielsweise im Common Lisp Object System CLOS vorfindet [Winston89, Steele90, Lawless91]), deren Auswahlmechanismus zumindest die *Typen* **aller** Funktionsargumente berücksichtigt.

Ebenso wie die Attribut- oder Konstanten-Deklarationen eines offenen Typs inkrementell erfolgen können, können auch die „Zweige“ einer stückweise definierten Funktion nach und nach an unterschiedlichen Stellen des Programmcodes formuliert und bei Bedarf ergänzt werden. Partielle Funktionen stellen somit das „prozedurale Pendant“ zu inkrementell definierten Datentypen dar und werden daher meist in Kombination mit diesen eingesetzt (vgl. § 4.6).

4.4.2.5 Sonstiges

Eine *Bit-shift-Operation* `x<<n` verschiebt das Bitmuster der Zahl `x` um `n` Positionen nach links. `1<<n` entspricht daher genau der Zweierpotenz von `n`, und `(1<<n) − 1` einem Bitmuster mit genau `n` Einsen.

Die Hilfsfunktion `select(w, i, b)`, die in der Funktion `testpar()` (Abb. 4.13) verwendet wird (und nichts mit der Sprache CH zu tun hat), selektiert aus dem Wort `w` diejenigen Aktionen (und fügt

sie zu einem neuen Teilwort zusammen), die an Positionen stehen, an denen das Bitmuster `i` den Wert `b` (0 oder 1) besitzt. Hierbei wird idealisierend angenommen, daß die „Breite“ des Bitmusters `i` mindestens so groß wie die Länge des Wortes `w` ist.

4.4.3 Implementierung der Iterationsoperatoren

Die Kompaktheit und Eleganz des Algorithmus resultiert nicht zuletzt aus der Tatsache, daß die in § 3.4.10 genannten Rekursionsgleichungen dazu verwendet werden, die Implementierung der beiden *Iterationsoperatoren* auf die Implementierung der zugehörigen *Kompositionsoperatoren* zurückzuführen. Um hierbei Endlosrekursionen zu vermeiden, muß der Parameter `start` der beiden Hilfsfunktionen `testseq()` und `testpar()` in diesem Fall mit dem Wert 1 versehen werden, um ein leeres Teilwort `u` zu verhindern.

Um zu zeigen, wie leicht sich die parallele Iteration bei dieser Vorgehensweise implementieren läßt, wurde sie *nicht* − wie in § 4.3.1 vorgeschlagen − auf einen äquivalenten Quantorausdruck zurückgeführt, sondern direkt implementiert. Aus diesem Grund wurde in Abb. 4.13 unterstellt, daß der Aufzählungstyp `Category` einen zusätzlichen Wert `ParIter` (parallele Iteration) besitzt, und daß der Rumpf des Ausdrucks ⊚ *y* ebenso wie bei der sequentiellen Iteration im Attribut `x/body` verfügbar ist.

4.4.4 Kritik

Obwohl der vorgestellte Algorithmus von Shaw außergewöhnlich kompakt und elegant ist, besitzt er den gravierenden Nachteil, daß er selbst für triviale parallele Kompositionen und Iterationen *exponentielle Laufzeit* bzgl. der Länge des Wortes `w` besitzt. Dies liegt darin begründet, daß die Hilfsfunktion `testpar()` im ungünstigsten Fall sämtliche 2^n möglichen „parallelen Zerlegungen“ des Wortes `w` (der Länge *n*) in Teilworte `u` und `v` durchlaufen muß, um festzustellen, ob das Wort `w` auf den Ausdruck *y* ⊚ *z* „paßt“. Bei Vorliegen einer parallelen Iteration wird die Funktion `testpar()` zusätzlich mit einer Verschachtelungstiefe der Größenordnung *n rekursiv* aufgerufen. Die hieraus resultierende Komplexität wird von Shaw selbst als indiskutabel eingestuft [Shaw80a].

Aber selbst wenn man auf die „bösartigen“ parallelen Operatoren verzichten würde, ist die Komplexität des Algorithmus nach wie vor inakzeptabel. Beispielsweise erhält man auch für eine einfache sequentielle Iteration eine Laufzeit der Größenordnung 2^n, wenn *n* wieder die Länge des Wortes `w` bezeichnet.[9] Im Gegensatz dazu lassen sich reguläre Ausdrücke (die man faktisch erhält, wenn man die parallelen Operatoren wegläßt; vgl. § 3.5.2) mit Hilfe endlicher Automaten bekanntlich mit *linearer* Komplexität implementieren.

4.4.5 Konsequenz

Ebenso wie man reguläre Ausdrücke üblicherweise nicht durch eine Eins-zu-eins-Umsetzung ihrer formalen Semantik, sondern durch eine Transformation in ein äquivalentes *Zustandsmodell* (in diesem Fall endliche Automaten) implementiert, muß man für eine effiziente Implementierung von Interaktionsausdrücken offenbar ebenfalls diesen (z. T. relativ „steinigen“) Umweg einschlagen.

Aus diesem Grund wird im folgenden Abschnitt 4.5 ein *operationales Modell* für Interaktionsausdrücke − bestehend aus (hierarchischen) *Zuständen*, *Zustandsprädikaten* und *Zustandsübergängen* − eingeführt, für das in Anhang B gezeigt wird, daß es äquivalent zur formalen Semantik gemäß § 3.3.4 ist. In § 4.6 wird dann die eigentliche Implementierung fortgesetzt, indem dieses Zustandsmodell

[9] Begründung: Bezeichnet man mit $f(n)$ die Gesamtzahl der rekursiven Aufrufe des Zweigs (`x/cat == Iter`) der Funktion `test(x, w)` (Abb. 4.12, unten) für ein Wort `w` der Länge *n*, so gilt für $n > 0$: Ein Aufruf von `test(x, w)` ruft die Hilfsfunktion `testseq()` auf, die *n*-mal rekursiv `test(x, v)` mit Worten `v` der Länge n-1, n-2, …, 0 aufruft. (Beachte: Der Parameter `z` der Funktion `testseq()` entspricht dem Parameter `x` der Funktion `test()`. Die Aufrufe `test(y, u)` werden nicht weiter betrachtet.) Somit gilt für $f(n)$ die Beziehung $f(n) = f(n-1) + f(n-2) + \ldots + f(0) + 1$. Ersetzt man hierin $f(n-2) + \ldots + f(0) + 1$ durch $f(n-1)$, was aufgrund derselben Formel für $n := n-1$ möglich ist, so erhält man $f(n) = f(n-1) + f(n-1) = 2 \cdot f(n-1)$. Zusammen mit dem Induktionsanfang $f(0) = 1$ ergibt sich hieraus die Behauptung $f(n) = 2^n$.

praktisch umgesetzt wird, und in § 4.7 wird gezeigt, daß sich der Umweg gelohnt hat, weil der resultierende Algorithmus für eine sehr große und praktisch relevante Teilmenge von Interaktionsausdrücken eine effiziente und damit praktisch verwendbare Implementierung darstellt.

4.5 Operationale Semantik von Interaktionsausdrücken

4.5.1 Einfache Zustandsmodelle

4.5.1.1 Basisfunktionen

Ein *einfaches Zustandsmodell* für Interaktionsausdrücke besteht aus einer Menge Θ von *Zuständen*, bei denen es sich prinzipiell um beliebige mathematische Objekte handeln kann, sowie den folgenden vier *Basisfunktionen*:

- Die *Initialisierungsfunktion* $\sigma : \Xi \to \Theta$ ordnet jedem Ausdruck $x \in \Xi$ einen *initialen Zustand* $\sigma(x) \in \Theta$ zu.
- Die *Zustandsübergangsfunktion* $\tau : \Theta \times \Sigma \to \Theta$ ordnet einem Zustand $s \in \Theta$ und einer konkreten Aktion $a \in \Sigma$ einen *transformierten Zustand* $s' = \tau_a(s) \in \Theta$ zu.
- Die *Zustandsprädikate* $\psi : \Theta \to \{ \top, \bot \}$ und $\varphi : \Theta \to \{ \top, \bot \}$ ordnen jedem Zustand $s \in \Theta$ zwei Wahrheitswerte $\psi(s)$ und $\varphi(s)$ zu, die mit $\top$ (wahr) bzw. $\bot$ (falsch) bezeichnet werden.
  Zustände, für die $\psi(s) = \top$ gilt, heißen *gültige Zustände*, Zustände mit $\varphi(s) = \top$ *Endzustände*.

Anmerkung: σ (**s**igma) steht für initial **s**tate, τ (**t**au) für **t**ransition. Die Prädikate ψ und φ stehen in direkter Beziehung zu den Mengen Ψ und Φ (vgl. § 4.5.1.3). $\top$ steht für **t**rue, das gespiegelte $\bot$ entsprechend für das Gegenteil false. Dem üblichen Stil mathematischer Arbeiten folgend, werden diese Kurzbezeichnungen (wie bereits in Kapitel 3) sprechenderen Namen wie z. B. state, trans oder true vorgezogen, um Formeln nicht unnötig aufzublähen.

4.5.1.2 Abgeleitete Funktionen

Aufbauend auf diesen Basisfunktionen, können die folgenden *abgeleiteten Funktionen* eines Zustandsmodells definiert werden:

- Durch mehrfache Anwendung der Zustandsübergangsfunktion τ erhält man zunächst die *Folgezustände* $\tau_w(s)$ eines beliebigen Zustands s:

  $$\tau_w(s) = \begin{cases} s & \text{für } w = \langle\rangle, \\ \tau_a\big(\tau_{\tilde{w}}(s)\big) & \text{für } w = \tilde{w}\,\langle a\rangle \text{ mit } \tilde{w} \in \Sigma^* \text{ und } a \in \Sigma, \end{cases}$$

  mit den zugehörigen Prädikaten

  $$\psi_w(s) = \psi\big(\tau_w(s)\big) \quad \text{und} \quad \varphi_w(s) = \varphi\big(\tau_w(s)\big).$$

- Für einen initialen Zustand $s = \sigma(x)$ erhält man speziell die Folgezustände $\sigma_w(x)$ des Ausdrucks x:

  $$\sigma_w(x) = \tau_w(\sigma(x)),$$

  die die Prädikate

  $$\psi_w(x) = \psi\big(\sigma_w(x)\big) = \psi\big(\tau_w(\sigma(x))\big) \quad \text{und} \quad \varphi_w(x) = \varphi\big(\sigma_w(x)\big) = \varphi\big(\tau_w(\sigma(x))\big)$$

  besitzen.

Die Menge aller möglichen Zustände eines Ausdrucks x wird mit $\Theta(x)$ bezeichnet:

$$\Theta(x) = \{ \sigma_w(x) \mid w \in \Sigma^* \}.$$

4.5.1.3 Korrektheitskriterium

Ein Zustandsmodell für Interaktionsausdrücke heißt *korrekt*, wenn für jeden Ausdruck $x \in \Xi$ und jedes Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) \Leftrightarrow \psi_w(x) = \top \quad \text{und} \quad w \in \Phi(x) \Leftrightarrow \varphi_w(x) = \top,$$

d. h. wenn man an den Prädikaten ψ und φ eines Zustands $\sigma_w(x)$ ablesen kann, ob w ein partielles bzw. vollständiges Wort des Ausdrucks x darstellt oder nicht.

Anmerkung: In einem korrekten Zustandsmodell gelten für jedes Wort $w \in \Sigma^*$ die Implikationen:

$$\varphi_w(x) = \top \quad \Rightarrow \quad w \in \Phi(x) \quad \Rightarrow \quad w \in \Psi(x) \quad \Rightarrow \quad \psi_w(x) = \top \qquad \text{(vgl. § 3.4.4)},$$

d. h. jeder Endzustand s eines Ausdrucks x ($\varphi(s) = \top$) ist auch ein gültiger Zustand ($\psi(s) = \top$).

Für Worte $w = \tilde{w}\,\langle a \rangle$ gilt außerdem:

$$\psi_w(x) = \top \quad \Rightarrow \quad w \in \Psi(x) \quad \Rightarrow \quad \tilde{w} \in \Psi(x) \quad \Rightarrow \quad \psi_{\tilde{w}}(x) = \top,$$

bzw. umgekehrt:

$$\psi_{\tilde{w}}(x) = \bot \quad \Rightarrow \quad \psi_w(x) = \bot,$$

d. h. die Folgezustände eines ungültigen Zustands ($\psi(s) = \bot$) sind ebenfalls ungültig.

4.5.1.4 Anschauliche Interpretation

Interpretiert man einen Interaktionsausdruck x wie in Kapitel 2 als Graph, so beschreibt sein initialer Zustand $\sigma(x)$ die *Ausgangssituation* beim Durchlaufen dieses Graphen, während ein Zustandsübergang $s' = \tau_a(s)$ dem *Durchlaufen* einer Aktion a entspricht.

Ein gültiger Folgezustand $\sigma_w(x)$ beschreibt die *Menge aller möglichen Positionen*, an denen sich ein Läufer (bzw. eine Mannschaft von Läufern) nach Durchlaufen der Aktionsfolge w befinden *könnte*, während ein ungültiger Zustand $\sigma_w(x)$ anzeigt, daß die Aktionsfolge w aus Sicht des Graphen unzulässig ist.

Ein Zustand $\sigma_w(x)$ ist genau dann ein Endzustand, wenn man nach Durchlaufen der Aktionsfolge w das *Ende* des Graphen erreicht haben *könnte*.

4.5.2 Optimierte Zustandsmodelle

4.5.2.1 Äquivalenz von Zuständen

Betrachtet man Zustände als Ausprägungen eines *abstrakten Datentyps*, deren interne Struktur *verborgen* ist, so stellen die Prädikate ψ und φ die einzigen *Methoden* (in objektorientierter Terminologie) dar, mit denen man *direkt* Aussagen über einen Zustand erhält. Unter Zuhilfenahme der Zustandsübergangsfunktion τ kann man jedoch auch *indirekt* Informationen über einen Zustand s erhalten, indem man die Prädikate ψ und φ auf seine Folgezustände $\tau_w(s)$ anwendet.

Wenn nun zwei Zustände s und $\hat{s}$ durch ausschließliche Anwendung der Funktionen ψ, φ und τ *nicht unterschieden* werden können, d. h. wenn

$$\psi_w(s) = \psi_w(\hat{s}) \quad \text{und} \quad \varphi_w(s) = \varphi_w(\hat{s}) \quad \text{für alle } w \in \Sigma^*$$

gilt, so heißen s und $\hat{s}$ *äquivalent* (engl. *equivalent* oder *bisimilar*), in Zeichen $s \cong \hat{s}$.

Da die Korrektheit eines Zustandsmodells lediglich von den „extern sichtbaren“ Werten der Prädikate $\psi_w(x)$ und $\varphi_w(x)$, nicht jedoch von der internen Struktur der zugehörigen Zustände $\sigma_w(x)$ abhängt, kann man einen Zustand s jederzeit durch einen äquivalenten Zustand $\hat{s}$ ersetzen, ohne dadurch die Korrektheit des Zustandsmodells zu verändern. Auf diese Weise kann ein Zustandsmodell *opti-*

miert werden, indem man Zustände s, für die die Berechnung der Funktionen $\psi(s)$, $\varphi(s)$ oder $\tau_a(s)$ aufwendig ist, durch äquivalente Zustände ersetzt, die sich effizienter verarbeiten lassen.

4.5.2.2 Optimierungsfunktion

Eine *Optimierungsfunktion* $\rho: \Theta \to \Theta$ ist eine Funktion, die jedem Zustand s einen *optimierten Zustand* $\hat{s} = \rho(s)$ zuordnet. ρ heißt *prädikattreu* oder *transparent*, wenn für jeden Zustand $s \in \Theta$ gilt:

$$\rho(s) \cong s.$$

Erweitert man ein einfaches Zustandsmodell um eine Optimierungsfunktion ρ, so erhält man ein *optimiertes Zustandsmodell* für Interaktionsausdrücke.

Anmerkung: ρ (**r**ho) steht für **r**educe.

4.5.2.3 Abgeleitete Funktionen

Kombiniert man die Funktionen τ und ρ, indem man sie nacheinander auf einen Zustand s anwendet, so erhält man die *optimierte Zustandsübergangsfunktion* $\hat{\tau}$:

$$\hat{\tau}_a(s) = \rho\big(\tau_a(s)\big),$$

die sich analog zu τ auf Worte $w \in \Sigma^*$ verallgemeinern läßt:

$$\hat{\tau}_w(s) = \begin{cases} s & \text{für } w = \langle\rangle, \\ \hat{\tau}_a\big(\hat{\tau}_{\tilde{w}}(s)\big) & \text{für } w = \tilde{w}\,\langle a\rangle \text{ mit } \tilde{w} \in \Sigma^* \text{ und } a \in \Sigma. \end{cases}$$

Wendet man diese Funktion auf einen initialen Zustand $s = \sigma(x)$ an, so erhält man entsprechend die *optimierten Folgezustände* des Ausdrucks x:

$$\hat{\sigma}_w(x) = \hat{\tau}_w(\sigma(x)) \qquad \text{für } w \in \Sigma^*,$$

mit den zugehörigen Prädikaten

$$\hat{\psi}_w(x) = \psi\big(\hat{\sigma}_w(x)\big) = \psi\big(\hat{\tau}_w(\sigma(x))\big) \quad \text{und} \quad \hat{\varphi}_w(x) = \varphi\big(\hat{\sigma}_w(x)\big) = \varphi\big(\hat{\tau}_w(\sigma(x))\big).$$

Die Menge aller optimierten Folgezustände eines Ausdrucks x wird mit $\hat{\Theta}(x)$ bezeichnet:

$$\hat{\Theta}(x) = \{\, \hat{\sigma}_w(x) \mid w \in \Sigma^* \,\}.$$

4.5.2.4 Korrektheitskriterium

Analog zu einem einfachen Zustandsmodell, heißt ein optimiertes Zustandsmodell für Interaktionsausdrücke *korrekt*, wenn für jeden Ausdruck $x \in \Xi$ und jedes Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) \Leftrightarrow \hat{\psi}_w(x) = \top \quad \text{und} \quad w \in \Phi(x) \Leftrightarrow \hat{\varphi}_w(x) = \top,$$

d. h. wenn man an den Prädikaten ψ und φ eines optimierten Zustands $\hat{\sigma}_w(x)$ ebenfalls ablesen kann, ob w ein partielles bzw. vollständiges Wort des Ausdrucks x darstellt oder nicht.

4.5.2.5 Satz

Ein optimiertes Zustandsmodell, bestehend aus den Funktionen σ, τ, ψ, φ und ρ, ist korrekt, wenn das zugehörige einfache Modell (bestehend aus den Funktionen σ, τ, ψ und φ) korrekt und die Optimierungsfunktion ρ transparent ist.

Anmerkung: Diese Aussage geht über die oben getroffene Feststellung hinaus, daß Zustände nach Belieben durch äquivalente Zustände ersetzt werden dürfen, ohne dadurch die Korrektheit des Zustandsmodells zu verändern. In einem optimierten Zustandsmodell werden nämlich nicht nur Zustände

durch äquivalente Zustände ersetzt, es wird auch eine modifizierte *Zustandsübergangsfunktion* $\hat{\tau}$ verwendet, obwohl die Äquivalenz von Zuständen nach wie vor durch die *einfache* Zustandsübergangsfunktion τ definiert wird.

Der Beweis dieses Satzes findet sich − ebenso wie alle übrigen Beweise des Abschnitts 4.5 − in Anhang B (§ B.3.1.3).

4.5.3 Konstruktion eines optimierten Zustandsmodells

4.5.3.1 Stückweise Definition der Basisfunktionen

In den nachfolgenden Abschnitten wird schrittweise ein optimiertes Zustandsmodell für Interaktionsausdrücke konstruiert, indem die Basisfunktionen σ, τ, ψ und φ sowie die Optimierungsfunktion ρ sukzessive für beliebige Ausdrücke definiert werden. Aus Gründen der Modularität und Übersichtlichkeit wird in jedem Teilabschnitt jedoch nur eine *Kategorie* von Ausdrücken (wie z. B. atomare Ausdrücke oder Disjunktionen) und zugehörigen Zuständen betrachtet, was zur Folge hat, daß die genannten Funktionen *stückweise* definiert werden.

Beispielsweise enthält § 4.5.4.1 die Definition von $\sigma(x)$ für atomare Ausdrücke $x \equiv a$, während in § 4.5.4.2 die entsprechende Definition für Disjunktionen $x \equiv y \bigcirc z$ gegeben wird, usw. Eine vollständige Definition von $\sigma(x)$ für beliebige Ausdrücke x würde man daher erhalten, wenn man alle diese Teildefinitionen zusammenfügt:

$$\sigma(x) = \begin{cases} \textit{siehe § 4.5.4.1} & \text{für } x \equiv a, \\ \textit{siehe § 4.5.4.2} & \text{für } x \equiv y \bigcirc z, \\ \vdots & \vdots \end{cases}$$

Ebenso wird in § 4.5.4.1 nur derjenige Teil der Zustandsübergangsfunktion τ definiert, der sich auf Zustände eines atomaren Ausdrucks bezieht (die entsprechend auch atomare Zustände genannt werden), während in § 4.5.4.2 nur Zustände von Disjunktionen (d. h. disjunktive Zustände) berücksichtigt werden, usw. Auch hier würde man eine vollständige Definition der Funktion τ erhalten, wenn man alle diese Teildefinitionen zusammenfügt:

$$\tau_a(s) = \begin{cases} \textit{siehe § 4.5.4.1} & \text{für einen atomaren Zustand } s, \\ \textit{siehe § 4.5.4.2} & \text{für einen disjunktiven Zustand } s, \\ \vdots & \vdots \end{cases}$$

In ähnlicher Weise verhält es sich auch mit den Funktionen ψ, φ und ρ.

4.5.3.2 Partielle Definition der Optimierungsfunktion

Zur Vereinfachung der Darstellung wird die Optimierungsfunktion ρ nicht direkt, sondern als Verkettung (Hintereinanderausführung) dreier Funktionen ρ_1, ρ_2 und ρ_3 definiert, die unterschiedliche Optimierungsschritte ausführen:

ρ_1 eliminiert ggf. ungültige Teilzustände[10] eines Zustands s (vgl. z. B. § 4.5.4.6);

ρ_2 ersetzt einen Zustand s ggf. durch einen seiner Teilzustände (dto.);

ρ_3 ersetzt einen ungültigen Zustand s durch den speziellen Zustand $\perp$.[11]

Zur weiteren Vereinfachung werden die Funktionen $\rho_i\colon \Theta \to \Theta$ lediglich *partiell* definiert und an un-

[10] Ein Zustand s eines zusammengesetzten Ausdrucks x besteht normalerweise aus Zuständen t seiner Teilausdrücke, die als *Teilzustände* von s bezeichnet werden.

[11] Dieser Zustand wird in § 4.5.4.1 eingeführt.

definierten Stellen $s \in \Theta$ durch die identische Funktion ergänzt. Formal ergibt sich die Optimierungsfunktion ρ daher wie folgt:

$$\rho(s) = \rho_3'\big(\rho_2'\big(\rho_1'(s)\big)\big) \quad \text{mit} \quad \rho_i'(s) = \begin{cases} \rho_i(s), & \text{falls definiert,} \\ s & \text{sonst,} \end{cases} \quad \text{für } i = 1, 2, 3.$$

Die Funktionen ρ_1 und ρ_2 werden in den nachfolgenden Abschnitten je nach Ausdruckskategorie geeignet definiert, während ρ_3 global wie folgt definiert ist:

$$\rho_3(s) = \bot, \quad \text{falls } \psi(s) = \bot.$$

4.5.3.3 Verifikation des Modells

Um die Korrektheit des *einfachen* Zustandsmodells (ohne die Optimierungsfunktion ρ) zu beweisen, müssen im Anschluß an die Definition der Basisfunktionen zunächst Aussagen über die Beschaffenheit der Folgezustände $\sigma_w(x)$ sowie der zugehörigen Prädikate $\psi_w(x)$ und $\varphi_w(x)$ formuliert und verifiziert werden, bevor das in § 4.5.1.3 genannte Korrektheitskriterium bewiesen werden kann. Da die entsprechenden Beweisschritte z. T. relativ umfangreich (und auch nicht sonderlich interessant) sind, wurden sie komplett in Anhang B (§ B.2.3 und § B.2.4) ausgelagert.

Gemäß § 4.5.2.5 muß anschließend noch die Transparenz der Optimierungsfunktion ρ nachgewiesen werden, um die Korrektheit des *optimierten* Zustandsmodells zu zeigen. Auch dieser Nachweis findet sich in Anhang B (§ B.3.2).

4.5.4 Definitionen für elementare Ausdrücke

4.5.4.1 Atomare Ausdrücke

Ein *atomarer Zustand*, d. h. ein Zustand eines atomaren Ausdrucks, ist entweder eine abstrakte Aktion $b \in \Gamma$ oder einer der Wahrheitswerte $\top$ oder $\bot$.

Ein Zustand $b \in \Gamma$ ist *gültig*, aber *kein Endzustand*:

$$\psi(b) = \top, \quad \varphi(b) = \bot.$$

Der Zustand $\top$ ist ein *Endzustand* (und daher auch ein gültiger Zustand), während der Zustand $\bot$ *ungültig* (und somit auch kein Endzustand) ist:

$$\psi(\top) = \varphi(\top) = \top, \quad \psi(\bot) = \varphi(\bot) = \bot.$$

Beim *Zustandsübergang* geht ein Zustand $b \in \Gamma$ in einen der Zustände $\top$ oder $\bot$ über, je nachdem ob die vorliegende Aktion a mit der Aktion b übereinstimmt oder nicht:

$$\tau_a(b) = (a = b) = \begin{cases} \top & \text{für } a = b, \\ \bot & \text{sonst.} \end{cases}$$

Die speziellen Zustände $\top$ und $\bot$ gehen bei jedem Zustandsübergang in den Zustand $\bot$ über:

$$\tau_a(\top) = \tau_a(\bot) = \bot \quad \text{für alle } a \in \Sigma.$$

Der *initiale Zustand* eines atomaren Ausdrucks $x \equiv b$ ist gleich der Aktion b:

$$\sigma(x) = b.$$

4.5.4.2 Disjunktion

Ein *disjunktiver Zustand*, d. h. ein Zustand einer Disjunktion $x \equiv y \bigcirc z$, ist ein 3-Tupel $s = [\bigcirc, l, r]$ mit einem Zustand l des linken Teilausdrucks y und einem Zustand r des rechten Teilausdrucks z, die als *Teilzustände* des Zustands s bezeichnet werden.

Ein solcher Zustand ist genau dann ein *gültiger Zustand* bzw. ein *Endzustand*, wenn er mindestens einen derartigen Teilzustand besitzt:[12]

$$\psi(s) = \psi(l) \vee \psi(r), \qquad \varphi(s) = \varphi(l) \vee \varphi(r).$$

Die *Zustandsübergangsfunktion* τ wird quasi rekursiv auf die Teilzustände l und r angewandt:

$$\tau_a(s) = [\bigcirc, \tau_a(l), \tau_a(r)].$$

Ein disjunktiver Zustand kann *optimiert* werden, wenn einer seiner Teilzustände ungültig ist:

$$\rho_2(s) = \begin{cases} l, & \text{falls } \psi(r) = \bot, \\ r, & \text{falls } \psi(l) = \bot. \end{cases}$$

(Wenn die Teilzustände l und r beide ungültig sind, ist es gleichgültig, ob man $\rho_2(s) = l$ oder $\rho_2(s) = r$ definiert.)

Beim *initialen Zustand* einer Disjunktion $x \equiv y \bigcirc z$ entsprechen die Teilzustände l und r den initialen Zuständen der Teilausdrücke y und z:

$$\sigma(x) = [\bigcirc, \sigma(y), \sigma(z)].$$

4.5.4.3 Option

Eine Option $x \equiv \circlearrowleft y$ wird als spezielle Disjunktion $x = y \bigcirc \varepsilon$ implementiert, wobei der initiale Zustand des *leeren Ausdrucks* ε wie folgt definiert ist:

$$\sigma(\varepsilon) = \top.$$

4.5.4.4 Konjunktion

Ein *konjunktiver Zustand*, d. h. ein Zustand einer Konjunktion $x \equiv y \bullet z$, ist ein 3-Tupel $s = [\bullet, l, r]$ mit einem Zustand l des linken Teilausdrucks y und einem Zustand r des rechten Teilausdrucks z, die − wie bei einem disjunktiven Zustand − als Teilzustände des Zustands s bezeichnet werden.

Die Definitionen der Funktionen ψ, φ, τ und σ sind vollkommen analog bzw. *dual* zu denen der Disjunktion:

$$\psi(s) = \psi(l) \wedge \psi(r), \qquad \varphi(s) = \varphi(l) \wedge \varphi(r),$$

$$\tau_a(s) = [\bullet, \tau_a(l), \tau_a(r)],$$

$$\sigma(x) = [\bullet, \sigma(y), \sigma(z)].$$

4.5.4.5 Synchronisation

Ein *Synchronisations-Zustand*, d. h. ein Zustand einer Synchronisation $x \equiv y \mathbin{\boldsymbol{\circ}} z$, ist ein 5-Tupel $s = [\boldsymbol{\circ}, y, z, l, r]$ mit einem Zustand l des linken Teilausdrucks y und einem Zustand r des rechten Teilausdrucks z. Neben diesen beiden Teilzuständen enthält ein Synchronisations-Zustand auch die Teilausdrücke y und z des Ausdrucks x, deren Alphabet für die Definition des Zustandsübergangs benötigt wird.[13]

[12] Trotz teilweise anderslautender Gepflogenheiten, ist es im folgenden günstig zu vereinbaren, daß die logischen Verknüpfungen $\vee$ und $\wedge$ *stärker* binden als das Gleichheitszeichen.

[13] Alternativ könnte man auch direkt die Alphabete $\alpha(y)$ und $\alpha(z)$ in den Zustand aufnehmen. Um mit späteren Zustandsdefinitionen (insbesondere Quantorzuständen, vgl. § 4.5.7) konsistent zu sein, werden jedoch die Teilausdrücke bevorzugt.

Ebenso wie bei der Konjunktion, ist ein solcher Zustand genau dann ein *gültiger Zustand* bzw. ein *Endzustand*, wenn beide Teilzustände die entsprechende Eigenschaft besitzen:

$$\psi(s) = \psi(l) \wedge \psi(r), \qquad \varphi(s) = \varphi(l) \wedge \varphi(r).$$

Beim *Zustandsübergang* muß berücksichtigt werden, ob die vorliegende Aktion a zum Alphabet des linken und/oder rechten Teilausdrucks der Synchronisation gehört:

$$\tau_a(s) = [\circ, y, z, l', r'] \quad \text{mit}$$

$$l' = \begin{cases} l & \text{für } a \in R := \alpha(z) \setminus \alpha(y), \\ \tau_a(l) & \text{sonst,} \end{cases} \qquad r' = \begin{cases} r & \text{für } a \in L := \alpha(y) \setminus \alpha(z), \\ \tau_a(r) & \text{sonst.} \end{cases}$$

Durch diese Definitionen werden im Prinzip vier verschiedene Fälle unterschieden:

1. Wenn a in *beiden* Teilausdrücken der Synchronisation auftritt, ist es in *keiner* der Differenzmengen L und R enthalten, so daß in diesem Fall für *beide* Teilzustände l und r ein Zustandsübergang durchgeführt wird.
   Dies entspricht der anschaulichen Definition, daß eine solche Aktion in *beiden* Zweigen der Synchronisation *gleichzeitig* durchlaufen werden muß.

2. Wenn a nur im *linken* Teilausdruck y auftritt, gehört es zwar zur Menge L, nicht jedoch zur Menge R, so daß in diesem Fall nur ein Zustandsübergang für den *linken* Teilzustand l durchgeführt wird, während der rechte Teilzustand r unverändert übernommen wird.
   Dies entspricht ebenfalls der anschaulichen Definition, daß eine solche Aktion nur im *linken* (d. h. oberen) Zweig der Synchronisation durchlaufen werden muß.

3. Wenn a nur im *rechten* Teilausdruck z auftritt, verhält es sich gerade umgekehrt: In diesem Fall ist a ein Element von R, nicht jedoch von L, so daß nur ein Zustandsübergang für den *rechten* Teilzustand r durchgeführt wird, während der linke Teilzustand l unverändert übernommen wird.
   Auch dies entspricht offensichtlich der anschaulichen Definition.

4. Wenn a in *keinem* Teilausdruck der Synchronisation auftritt, gehört es (wie in Fall 1) *weder* zur Menge L *noch* zur Menge R, d. h. in diesem Fall wird *sowohl* ein Zustandsübergang für den linken Teilzustand l *als auch* für den rechten Teilzustand r durchgeführt. Da die Aktion a jedoch − unter der Voraussetzung eines korrekten Zustandsmodells − *weder* vom Zustand l *noch* vom Zustand r akzeptiert werden wird, werden die resultierenden Zustände l' und r' beide *ungültig* sein.
   Zusammen mit den Definitionen der Prädikate ψ und φ entspricht daher auch dieser Fall der anschaulichen Definition, nach der eine solche Aktion *überhaupt nicht* durchlaufen werden darf.

Beim *initialen Zustand* einer Synchronisation $x \equiv y \circ z$ entsprechen die Teilzustände l und r den initialen Zuständen der Teilausdrücke y und z:

$$\sigma(x) = [\circ, y, z, \sigma(y), \sigma(z)].$$

4.5.4.6 Sequentielle Komposition

Ein *sequentieller Kompositions-Zustand*, d. h. ein Zustand einer sequentiellen Komposition $x \equiv y - z$, ist ein 4-Tupel $s = [-, z, l, R]$ mit einem Zustand l des linken Teilausdrucks y und einer Menge R von Zuständen des rechten Teilausdrucks z. Neben diesen Teilzuständen enthält ein sequentieller Kompositions-Zustand den rechten Teilausdruck z des Ausdrucks x, der für die Definition des Zustandsübergangs benötigt wird.

Ein solcher Zustand s ist genau dann *gültig*, wenn er mindestens einen gültigen Teilzustand besitzt. Er ist genau dann ein *Endzustand*, wenn die Menge R mindestens einen Endzustand enthält:

$$\psi(s) = \psi(l) \vee \bigvee_{r \in R} \psi(r), \qquad \varphi(s) = \bigvee_{r \in R} \varphi(r).$$

Die *Zustandsübergangsfunktion* $\tau_a(s)$ wird rekursiv auf die Teilzustände l und $r \in R$ angewandt. Außerdem wird die resultierende Menge R' immer dann um eine „neue Ausprägung" des initialen Zustands $\sigma(z)$ des rechten Teilausdrucks z erweitert, wenn der Zustand $l' = \tau_a(l)$ des linken Teilausdrucks y ein Endzustand ist, d. h. wenn die bis jetzt durchlaufene Aktionsfolge ein vollständiges Wort von y darstellt:[14]

$$\tau_a(s) = [-, z, l', R'] \quad \text{mit} \quad l' = \tau_a(l) \quad \text{und} \quad R' = \{\, \tau_a(r) \mid r \in R \,\} \cup \left\{\, \sigma(z) \;\middle|\; \varphi\!\left(l'\right) \right\}.$$

Ein sequentieller Kompositions-Zustand kann *optimiert* werden, indem zunächst ungültige Teilzustände $r \in R$ eliminiert werden:

$$\rho_1(s) = \left[-, z, l, \hat{R}\right] \quad \text{mit} \quad \hat{R} = \{\, r \in R \mid \psi(r) \,\}.$$

Anschließend kann (der hieraus resultierende Zustand) s u. U. durch einen einzelnen Teilzustand $r \in R$ ersetzt werden:

$$\rho_2(s) = r, \quad \text{falls } R = \{\, r \,\} \quad \text{und} \quad \psi(l) = \bot \text{ oder } l = \top.$$

Beim *initialen Zustand* einer sequentiellen Komposition $x \equiv y - z$ entspricht der Teilzustand l dem initialen Zustand von y, während die Menge R genau dann den initialen Zustand $\sigma(z)$ enthält, wenn l gleichzeitig ein Endzustand ist, d. h. wenn das leere Wort bereits ein *vollständiges* Wort von y darstellt:

$$\sigma(x) = [-, z, l, R] \quad \text{mit} \quad l = \sigma(y) \quad \text{und} \quad R = \{\, \sigma(z) \mid \varphi(l) \,\}.$$

Anmerkung: Die bedingte Erweiterung der Menge R bzw. R' um den Zustand $\sigma(z)$ spiegelt die Tatsache wider, daß man beim Durchlaufen des Graphen $y - z$ genau dann mit der Traversierung von z beginnen kann, wenn man y *vollständig* durchlaufen hat, d. h. einen Endzustand von y erreicht hat.

4.5.4.7 Sequentielle Iteration

Ein *sequentieller Iterations-Zustand*, d. h. ein Zustand einer sequentiellen Iteration $x \equiv \ominus\, y$, ist ein 3-Tupel $s = [\ominus, y, T]$ mit einer Menge T von Zuständen des Teilausdrucks y. Ähnlich wie ein sequentieller Kompositions-Zustand, enthält ein sequentieller Iterations-Zustand neben diesen Teilzuständen den Rumpf y des Ausdrucks x, der für die Definition des Zustandsübergangs benötigt wird.

Ein solcher Zustand ist genau dann ein *gültiger Zustand* bzw. ein *Endzustand*, wenn er mindestens einen entsprechenden Teilzustand besitzt:

$$\psi(s) = \bigvee_{t \in T} \psi(t), \qquad \varphi(s) = \bigvee_{t \in T} \varphi(t).$$

Ähnlich wie bei der sequentiellen Komposition, wird die *Zustandsübergangsfunktion* τ rekursiv auf die Teilzustände $t \in T$ angewandt und die resultierende Menge T' immer dann um eine „neue Ausprägung" des Zustands $\sigma(y)$ erweitert, wenn einer der Folgezustände $t' = \tau_a(t)$ ein Endzustand ist, d. h. wenn die bis jetzt durchlaufene Aktionsfolge ein vollständiges Wort des Ausdrucks x darstellt:

$$\tau_a(s) = [\ominus, y, T''] \quad \text{mit} \quad T' = \{\, \tau_a(t) \mid t \in T \,\} \quad \text{und} \quad T'' = T' \cup \left\{\, \sigma(y) \;\middle|\; \bigvee_{t' \in T'} \varphi\!\left(t'\right) \right\}.$$

Ebenfalls analog zur sequentiellen Komposition, kann ein sequentieller Iterations-Zustand *optimiert* werden, indem ungültige Teilzustände $t \in T$ eliminiert werden:

$$\rho_1(s) = \left[\ominus, y, \hat{T}\right] \quad \text{mit} \quad \hat{T} = \{\, t \in T \mid \psi(t) \,\}.$$

[14] Je nachdem, ob das Prädikat $\varphi(l')$ erfüllt ist oder nicht, entspricht die Menge $\left\{\, \sigma(z) \mid \varphi(l') \right\}$ entweder der einelementigen Menge $\{\, \sigma(z) \,\}$ oder aber der leeren Menge $\varnothing$.

Beim *initialen Zustand* einer sequentiellen Iteration $x \equiv \ominus\, y$ enthält die Menge T die Zustände $\sigma(y)$ und $\top$, wobei letzterer dafür sorgt, daß der Zustand gleichzeitig ein Endzustand ist:

$$\sigma(x) = [\ominus, y, T] \quad \text{mit} \quad T = \{\, \sigma(y), \top \,\}.$$

4.5.4.8 Parallele Komposition

Ein *paralleler Kompositions-Zustand*, d. h. ein Zustand einer parallelen Komposition $x \equiv y \circledcirc z$, ist ein 2-Tupel $s = [\circledcirc, A]$ mit einer Menge A von Zustandspaaren $[l, r]$, die als *Alternativen* des Zustands bezeichnet werden. Jede Alternative $[l, r] \in A$ besteht aus einem Zustand l des linken Teilausdrucks y und einem Zustand r des rechten Teilausdrucks z.

Ein solcher Zustand ist genau dann ein *gültiger Zustand* bzw. ein *Endzustand*, wenn er mindestens eine Alternative $[l, r]$ enthält, deren Zustände l und r *beide* gültige Zustände bzw. Endzustände sind:

$$\psi(s) = \bigvee_{[l, r] \in A} (\psi(l) \wedge \psi(r)), \qquad \varphi(s) = \bigvee_{[l, r] \in A} (\varphi(l) \wedge \varphi(r)).$$

Da eine Aktion a potentiell von jedem der beiden Teilausdrücke verarbeitet werden kann, wird beim *Zustandsübergang* jede Alternative $[l, r]$ durch zwei neue Alternativen $[l', r]$ und $[l, r']$ mit $l' = \tau_a(l)$ und $r' = \tau_a(r)$ ersetzt:

$$\tau_a(s) = [\circledcirc, A'] \quad \text{mit} \quad A' = \{\, [\tau_a(l), r] \mid [l, r] \in A \,\} \cup \{\, [l, \tau_a(r)] \mid [l, r] \in A \,\}.$$

Ein paralleler Kompositions-Zustand kann *optimiert* werden, indem Alternativen mit ungültigen Teilzuständen eliminiert werden:

$$\rho_1(s) = \left[\circledcirc, \hat{A}\right] \quad \text{mit} \quad \hat{A} = \{\, [l, r] \in A \mid \psi(l) \wedge \psi(r) \,\}.$$

Der *initiale Zustand* einer parallelen Komposition $x \equiv y \circledcirc z$ besitzt genau eine Alternative, die aus den initialen Zuständen der Teilausdrücke y und z besteht:

$$\sigma(x) = [\circledcirc, A] \quad \text{mit} \quad A = \{\, [\sigma(y), \sigma(z)] \,\}.$$

4.5.4.9 Parallele Iteration

Da eine parallele Iteration, wie bereits erwähnt (§ 4.3.1), durch einen parallelen Quantorausdruck ersetzt wird, kann auf eine separate operationale Definition dieses Operators verzichtet werden.

4.5.5 Definitionen für Multiplikatorausdrücke

Multiplikatorausdrücke können grundsätzlich gemäß ihrer Definition auf elementare Ausdrücke zurückgeführt werden (vgl. § 3.3.2.1), so daß eine separate operationale Definition prinzipiell ebenfalls entbehrlich ist. Da die hieraus resultierende Implementierung aber − insbesondere im Kontext paralleler Multiplikatoren $\overset{n}{\circledcirc}$ − vergleichsweise ineffizient wäre, ist es ratsam, für diese Operatoren spezialisierte Definitionen zu entwickeln, die die Tatsache berücksichtigen, daß bei einem Multiplikatorausdruck mehrmals *derselbe* Teilausdruck mit sich selbst verknüpft wird. Aus Platzgründen wird auf die entsprechenden Definitionen jedoch nicht näher eingegangen.

4.5.6 Vorbereitungen für Quantorausdrücke

4.5.6.1 Konkretisierte Zustände

Analog zu konkretisierten Aktionen und Ausdrücken (vgl. § 3.3.3.1), erhält man einen *konkretisierten Zustand* s_p^ω (für $s \in \Theta$, $p \in \Pi$ und $\omega \in \Omega$), indem man jede Aktion a, die in s oder einem seiner Teil-

zustände auftritt, durch die konkretisierte Aktion a_p^ω ersetzt, d. h. indem man den „Konkretisierungsoperator" $(\ldots)_p^\omega$ rekursiv auf alle Komponenten des Zustands s anwendet:

$$s_p^\omega = \begin{cases} s & \text{für } s \in \{\top, \bot\}, \\ a_p^\omega & \text{für } s = a, \\ \left[\bigcirc, l_p^\omega, r_p^\omega\right] & \text{für } s = [\bigcirc, l, r], \\ \left[\bullet, l_p^\omega, r_p^\omega\right] & \text{für } s = [\bullet, l, r], \\ \left[\boldsymbol{\circ}, y_p^\omega, z_p^\omega, l_p^\omega, r_p^\omega\right] & \text{für } s = [\boldsymbol{\circ}, y, z, l, r], \\ \left[-, z_p^\omega, l_p^\omega, R_p^\omega\right] & \text{für } s = [-, z, l, R], \\ \left[\ominus, y_p^\omega, T_p^\omega\right] & \text{für } s = [\ominus, y, T], \\ \left[\circledcirc, A_p^\omega\right] & \text{für } s = [\circledcirc, A]. \end{cases}$$

Hierfür sei außerdem

$$R_p^\omega = \left\{ r_p^\omega \;\middle|\; r \in R \right\}, \quad T_p^\omega = \left\{ t_p^\omega \;\middle|\; t \in T \right\} \quad \text{und} \quad A_p^\omega = \left\{ \left[l_p^\omega, r_p^\omega\right] \;\middle|\; [l, r] \in A \right\}$$

definiert.

Für die in § 4.5.7 definierten Zustände von Quantorausdrücken kann die Definition von s_p^ω in naheliegender Weise verallgemeinert werden.

Den initialen Zustand $\sigma\left(x_p^\omega\right)$ eines konkretisierten Ausdrucks x_p^ω erhält man offensichtlich, indem man den initialen Zustand $\sigma(x)$ des ursprünglichen Ausdrucks x konkretisiert, d. h. die Operationen „initialer Zustand" und „Konkretisierung" sind vertauschbar:

$$\sigma\left(x_p^\omega\right) = (\sigma(x))_p^\omega .$$

4.5.6.2 Relevante Parameterwerte

Wie in § 3.3.3.1 erläutert, entspricht ein Quantorausdruck $\underset{p}{\bigcirc}\, y$ anschaulich einem *unendlichen* Ausdruck $\underset{\omega \in \Omega}{\bigcirc} y_p^\omega$. Bei Vorliegen einer konkreten Aktion $a \in \Sigma$ sind von den unendlich vielen Zweigen y_p^ω allerdings nur diejenigen wirklich interessant, die sich − was das Auftreten der Aktion a anbelangt − vom abstrakten Zweig y *unterscheiden*.

Beispiel

Beispielsweise sind für $a = \mathsf{abc(X, Y, Z)}$ mit $\mathsf{X, Y, Z} \in \Omega$ und

$$y \equiv \mathsf{abc}(p, \mathsf{Y, Z}) - \mathsf{abc}(\mathsf{X}, p, \mathsf{Z}) - \underline{\mathsf{abc(X, Y, Z)}} \qquad \text{mit } p \in \Pi$$

nur die Zweige

$$y_p^{\mathsf{X}} \equiv \underline{\mathsf{abc(X, Y, Z)}} - \mathsf{abc(X, X, Z)} - \underline{\mathsf{abc(X, Y, Z)}}$$

und

$$y_p^{\mathsf{Y}} \equiv \mathsf{abc(Y, Y, Z)} - \underline{\mathsf{abc(X, Y, Z)}} - \underline{\mathsf{abc(X, Y, Z)}},$$

nicht jedoch die Zweige

$$y_p^\omega \equiv \mathsf{abc}(\omega, \mathsf{Y, Z}) - \mathsf{abc}(\mathsf{X}, \omega, \mathsf{Z}) - \underline{\mathsf{abc(X, Y, Z)}} \qquad \text{für } \omega \notin \{\mathsf{X, Y}\}$$

interessant, da sich nur die beiden erstgenannten bzgl. des Auftretens der (jeweils unterstrichenen) Aktion a vom abstrakten Zweig y unterscheiden. Dies hat zur Folge, daß sich alle Zweige y_p^ω mit $\omega \notin \{\mathsf{X, Y}\}$, was die Verarbeitung der konkreten Aktion a anbelangt, genauso wie der abstrakte

Zweig y verhalten werden und dieser somit als *Stellvertreter* all dieser Zweige verwendet werden kann.[15] Folglich müssen bei der Verarbeitung von a nur noch die Zweige y_p^{X} und y_p^{Y} gesondert behandelt werden.

Allgemeines Prinzip

Um diese *relevanten* Zweige eines Quantorausdrucks $\underset{p}{\bigcirc}\, y$ − bzw. ihre zugehörigen Parameterwerte ω − allgemein zu bestimmen, kann man wie folgt vorgehen:

- Man entfernt aus dem Alphabet des Quantorrumpfs y die Aktion a (sofern vorhanden), weil sämtliche Ausprägungen dieser Aktion im abstrakten Zweig y in gleicher Weise (d. h. an denselben Stellen) auch in allen konkretisierten Zweigen y_p^{ω} auftreten und somit keinen Unterschied zwischen dem abstrakten Zweig y und einem konkretisierten Zweig y_p^{ω} darstellen können.
- Die verbleibende Menge $A = \alpha(y) \setminus \{ a \}$ wird nacheinander für jeden Wert $\omega \in \Omega$ konkretisiert. Wenn eine der resultierenden Mengen A_p^{ω} die Aktion a enthält, so bedeutet dies, daß der entsprechende konkretisierte Zweig y_p^{ω} die Aktion a an einer Stelle enthält, an der sie im abstrakten Zweig y *nicht* auftritt. (Der abstrakte Zweig enthält an dieser Stelle eine zu a *ähnliche* Aktion $\tilde{a}$, die erst durch die Konkretisierung $\tilde{a}_p^{\omega}$ in a übergeht; siehe unten.) Somit stellt der entsprechende Wert ω einen für die Verarbeitung der Aktion a relevanten Parameterwert des Quantorrumpfs y dar.

Definition

Formal kann die Menge der *relevanten Parameterwerte* der Aktion a bzgl. des Ausdrucks y und des Quantorparameters p wie folgt definiert werden:

$$\Omega_a(y,\, p) = \left\{ \omega \in \Omega \;\middle|\; a \in \big(\alpha(y) \setminus \{ a \}\big)_p^{\omega} \right\}.$$

Diese Definition läßt sich unmittelbar für Worte $w \in \Sigma^*$ verallgemeinern:

$$\Omega_w(y,\, p) = \bigcup_{i=1}^{n} \Omega_{w_i}(y,\, p) \qquad \text{für } w = \langle w_1, \ldots, w_n \rangle.$$

Anmerkung: Da die Menge $A = \alpha(y) \setminus \{ a \}$ die Aktion a nicht enthält, kann die konkretisierte Menge A_p^{ω} ebendiese Aktion nur enthalten, wenn A eine Aktion $\tilde{a}$ enthält, die durch die Konkretisierung $\tilde{a}_p^{\omega}$ gleich a wird. Da dies offensichtlich nur für einen Wert ω möglich ist, der in der Parameterliste von a vorkommt, gilt somit:

$$\Omega_a(y,\, p) \subseteq \{ a_1, \ldots, a_n \} \qquad \text{für } a = [a_0, a_1, \ldots, a_n].$$

Außerdem ist die Bedingung $\tilde{a}_p^{\omega} = a$ für ein $\tilde{a} \in A \subseteq \alpha(y)$ nur erfüllbar, wenn die *Breite* $|a|$ (d. h. die Anzahl der Parameter) der Aktion $a \in \Sigma$ kleiner oder gleich der *maximalen Breite* von Aktionen $\tilde{a} \in \alpha(y)$ ist, d. h. wenn $|a| \le k := \max_{\tilde{a} \in \alpha(y)} |\tilde{a}|$ gilt. Somit gilt für beliebige Aktionen $a = [a_0, a_1, \ldots, a_n] \in \Sigma$:

$$\begin{array}{ll} \Omega_a(y,\, p) \subseteq \{ a_1, \ldots, a_n \} & \text{für } n \le k, \\ \Omega_a(y,\, p) = \emptyset & \text{sonst,} \end{array}$$

und deshalb:

$$|\Omega_a(y,\, p)| \le k = \max_{\tilde{a} \in \alpha(y)} |\tilde{a}| \qquad \text{für alle } a \in \Sigma,$$

[15] Dieser Überlegung liegt die Beobachtung zugrunde, daß das Resultat eines Zustandsübergangs $\tau_a(s)$ für $s \in \Theta(y)$ letztlich nur von den *atomaren* Zustandsübergängen $\tau_a(b)$ für $b \in \alpha(y)$ abhängt, da alle anderen Zustandsübergänge die Aktion a rekursiv an ihre Teilzustände „weiterreichen“. Das Resultat eines atomaren Zustandsübergangs $\tau_a(b)$ wiederum hängt lediglich vom Wahrheitswert der Bedingung $a = b$, nicht jedoch von der exakten Beschaffenheit der Aktion b ab (vgl. § 4.5.4.1).

wobei die Konstante k *nicht* von der Aktion a, sondern nur vom Ausdruck y abhängt. Insbesondere ist die Menge $\Omega_a(y,\ p)$ stets *endlich*.

4.5.6.3 Substitutionsprinzip

Satz

Für einen Quantorausdruck $\underset{p}{\bigcirc}\ y$, ein Wort $w \in \Sigma^*$ und einen bzgl. y und p *irrelevanten* Parameterwert $\omega \notin \Omega_w(y,\ p)$ des Wortes w gilt:

$$\sigma_w\left(y_p^\omega\right) = \left(\sigma_w(y)\right)_p^\omega,$$

d. h. der Zustand $\sigma_w\left(y_p^\omega\right)$ des konkretisierten Zweigs y_p^ω nach Verarbeitung des Wortes w kann „nachträglich" durch Konkretisierung des entsprechenden Zustands $\sigma_w(y)$ des abstrakten Zweigs y gewonnen werden.

Erläuterung

Dieser Satz formalisiert die in § 4.5.6.2 formulierte Behauptung, daß der abstrakte Zweig y eines Quantorausdrucks als *Stellvertreter* aller irrelevanten Zweige y_p^ω (mit $\omega \notin \Omega_w(y,\ p)$) verwendet werden kann. Sobald ein Wert ω während der Verarbeitung einer Aktionsfolge $a_1, a_2, \ldots$ relevant wird (weil er zur Menge $\Omega_{a_i}(y,\ p)$ der als nächstes zu verarbeitenden Aktion a_i gehört), kann der Zustand seines Zweigs y_p^ω somit aus dem Zustand des abstrakten Zweigs y konstruiert und anschließend eigenständig weiterverarbeitet werden.

Das Substitutionsprinzip − zusammen mit der Beobachtung, daß die Menge der relevanten Parameterwerte stets endlich ist − stellt somit die formale Grundlage für die prinzipielle Implementierbarkeit von Quantorausdrücken dar, weil dadurch sichergestellt ist, daß ein konzeptuell *unendlicher* Ausdruck dennoch unter Verwendung *endlicher* Ressourcen (Speicherplatzbedarf und Rechenaufwand) verarbeitet werden kann.

Der Beweis des Satzes wird in Anhang B (§ B.2.3 und § B.2.4) zusammen mit dem Korrektheitsnachweis des einfachen Zustandsmodells erbracht.

4.5.6.4 Erweiterte Zustände

Ein *erweiterter Zustand* ist ein Zustand/Wert-Paar $[t,\ \omega]$, bestehend aus einem Zustand $t \in \Theta\left(y_p^\omega\right)$ des konkretisierten Zweigs y_p^ω und dem zugehörigen Wert $\omega \in \Omega$. Als Sonderfall werden außerdem Paare $[t,\ p]$ zugelassen, die aus einem Zustand $t \in \Theta(y)$ des abstrakten Zweigs y und dem Quantorparameter p (als „Pseudowert") bestehen. Dadurch können die Zustände des abstrakten Zweigs y später in gleicher Weise wie die der konkretisierten Zweige beschrieben und verarbeitet werden.

Aus Gründen der Bequemlichkeit werden die Funktionen τ, ψ und φ wie folgt auf erweiterte Zustände ausgedehnt:

$$\tau_a\left([t,\ \omega]\right) = [\tau_a(t),\ \omega], \qquad \psi\left([t,\ \omega]\right) = \psi(t) \qquad \text{und} \qquad \varphi\left([t,\ \omega]\right) = \varphi(t).$$

Projiziert man eine Menge T von erweiterten Zuständen auf die Menge Ω, so erhält man die Menge

$$\Omega(T) = \{\ \omega \in \Omega \mid \exists\, t \in \Theta\colon [t,\ \omega] \in T\ \}$$

aller Werte von T.

4.5.7 Definitionen für Quantorausdrücke

4.5.7.1 Disjunktions-Quantorausdrücke

Ein *disjunktiver Quantorzustand*, d. h. ein Zustand eines Disjunktions-Quantorausdrucks $x \equiv \bigcirc_p y$, ist ein 5-Tupel $s = [\circ, y, p, r, T]$ mit einem Zustand r des abstrakten Zweigs y und einer endlichen Menge T von erweiterten Zuständen gewisser konkretisierter Zweige y_p^ω. Neben diesen Teilzuständen enthält ein disjunktiver Quantorzustand den Rumpf y sowie den Parameter p des Ausdrucks x, die beide für die Definition des Zustandsübergangs benötigt werden.

Ein solcher Zustand ist genau dann ein *gültiger Zustand* bzw. ein *Endzustand*, wenn er mindestens einen entsprechenden Teilzustand besitzt:

$$\psi(s) = \psi(r) \vee \bigvee_{t \in T} \psi(t), \qquad \varphi(s) = \varphi(r) \vee \bigvee_{t \in T} \varphi(t).$$

Die *Zustandsübergangsfunktion* $\tau_a(s)$ wird rekursiv auf alle Teilzustände angewandt. Zuvor wird jedoch der Zustand r des abstrakten Zweigs y für jeden relevanten Parameterwert $\omega \in \Omega_a(y, p)$ der Aktion a, der in der Menge T (bzw. $\Omega(T)$) noch nicht enthalten ist, zu einem erweiterten Zustand $\left[r_p^\omega, \omega\right]$ konkretisiert:

$$\tau_a(s) = [\circ, y, p, r', T''] \quad \text{mit}$$

$$r' = \tau_a(r), \quad T' = T \cup \left\{ \left[r_p^\omega, \omega\right] \,\middle|\, \omega \in \Omega_a(y, p) \setminus \Omega(T) \right\} \quad \text{und} \quad T'' = \{\, \tau_a(t) \mid t \in T' \,\}.$$

Wenn der Zustand r ungültig ist, kann ein disjunktiver Quantorzustand *optimiert* werden, indem ungültige erweiterte Zustände $t \in T$ eliminiert werden:

$$\rho_1(s) = \left[\circ, y, p, \bot, \hat{T}\right] \quad \text{mit} \quad \hat{T} = \{\, t \in T \mid \psi(t) \,\}, \quad \text{falls } \psi(r) = \bot.$$

Wenn hierbei nur ein Zustand/Wert-Paar $[\top, \omega]$ verbleibt, kann der gesamte Zustand durch den Zustand $\top$ ersetzt werden:

$$\rho_2(s) = \top, \quad \text{falls } \psi(r) = \bot \text{ und } T = \{\, [\top, \omega] \,\} \text{ für ein } \omega \in \Omega.$$

Anmerkung: Durch das Entfernen ungültiger erweiterter Zustände $t \in T$ wird die beim Zustandsübergang relevante Menge $\Omega(T)$ verkleinert. Damit dies keine unerwünschten „Nebenwirkungen" hat, darf die Optimierung ρ_1 nur vorgenommen werden, wenn der Zustand r ungültig ist.

Die Optimierung ρ_2 könnte prinzipiell auch wie folgt verallgemeinert werden:

$$\rho_2(s) = t, \quad \text{falls } \psi(r) = \bot \text{ und } T = \{\, [t, \omega] \,\} \text{ für ein } \omega \in \Omega.$$

Für die implementierungstechnische Umsetzung (§ 4.6.4.1) ist dies jedoch insofern problematisch, als dort der Zustand t nur *zusammen* mit dem Wert ω korrekt weiterverarbeitet werden kann. Dies wiederum liegt darin begründet, daß Zustände in der Implementierung immer nur *temporär* konkretisiert werden und anschließend den ihnen zugeordneten Wert ω wieder „vergessen" (vgl. § 4.6.3.1). Daher wird ρ_2 nur für den Spezialfall $T = \{\, [\top, \omega] \,\}$ definiert.

Beim *initialen Zustand* eines Disjunktions-Quantorausdrucks $x \equiv \bigcirc_p y$ entspricht r dem initialen Zustand von y und T der leeren Menge:

$$\sigma(x) = [\circ, y, p, \sigma(y), \varnothing].$$

4.5.7.2 Konjunktions-Quantorausdrücke

Ein *konjunktiver Quantorzustand*, d. h. ein Zustand eines Konjunktions-Quantorausdrucks $x \equiv \bullet_p y$, ist ein 5-Tupel $s = [\bullet, y, p, r, T]$ mit y, p, r und T wie bei einem disjunktiven Quantorzustand.

Auch die folgenden Definitionen sind, ähnlich wie bei der elementaren Konjunktion, im wesentlichen analog bzw. dual zu denen von Disjunktions-Quantorausdrücken:

$$\psi(s) = \psi(r) \wedge \bigwedge_{t \in T} \psi(t), \qquad \varphi(s) = \varphi(r) \wedge \bigwedge_{t \in T} \varphi(t),$$

$$\tau_a(s) = [\bullet, y, p, r', T''] \quad \text{mit}$$

$$r' = \tau_a(r), \quad T' = T \cup \left\{ \left[r_p^\omega, \omega\right] \,\middle|\, \omega \in \Omega_a(y, p) \setminus \Omega(T) \right\} \quad \text{und} \quad T'' = \{ \tau_a(t) \mid t \in T' \},$$

$$\sigma(x) = [\bullet, y, p, \sigma(y), \varnothing].$$

4.5.7.3 Synchronisations-Quantorausdrücke

Ein *Synchronisations-Quantorzustand*, d. h. ein Zustand eines Synchronisations-Quantorausdrucks $x \equiv \underset{p}{\bigcirc}\, y$, ist ein 5-Tupel $s = [\circ, y, p, r, T]$ mit y, p, r und T wie bei einem disjunktiven oder konjunktiven Quantorzustand.

Auch die Prädikate ψ und φ entsprechen denen eines konjunktiven Quantorzustands:

$$\psi(s) = \psi(r) \wedge \bigwedge_{t \in T} \psi(t), \qquad \varphi(s) = \varphi(r) \wedge \bigwedge_{t \in T} \varphi(t).$$

Beim *Zustandsübergang* muß jedoch, ähnlich wie bei einer elementaren Synchronisation, das Alphabet der einzelnen Zweige berücksichtigt werden:

$$\tau_a(s) = [\circ, y, p, r', T''] \quad \text{mit}$$

$$r' = \vartheta_a^p(r), \quad T' = T \cup \left\{ \left[r_p^\omega, \omega\right] \,\middle|\, \omega \in \Omega_a(y, p) \setminus \Omega(T) \right\}, \quad T'' = \{ \vartheta_a^\omega(t) \mid t \in T' \}$$

und der Hilfsfunktion

$$\vartheta_a^\omega(t) = \begin{cases} t & \text{für } a \in \left(\bigcup_{\pi \in \Omega} A_p^\pi \right) \setminus A_p^\omega \text{ mit } A = \alpha(y), \\ \tau_a(t) & \text{sonst.} \end{cases}$$

Anmerkung: Die Menge

$$\bigcup_{\pi \in \Omega} A_p^\pi = \bigcup_{\pi \in \Omega} (\alpha(y))_p^\pi = \bigcup_{\pi \in \Omega} \alpha\left(y_p^\pi\right) = \alpha(x)$$

beschreibt das Alphabet des Gesamtausdrucks x, während die Menge $A_p^\omega = (\alpha(y))_p^\omega = \alpha\left(y_p^\omega\right)$ dem Alphabet des Zweigs y_p^ω entspricht. Ihre Differenzmenge enthält daher all jene Aktionen des Ausdrucks x, die *nicht* im betrachteten Zweig y_p^ω vorkommen, und entspricht somit der Menge $\kappa_x\left(y_p^\omega\right)$ aus § 3.3.3.2.

Die Menge A_p^p, die beim Aufruf der Hilfsfunktion $r' = \vartheta_a^p(r)$ auftritt, sei identisch zur Menge A definiert. (Da der Parameter p quasi durch sich selbst ersetzt wird, bleibt er faktisch unverändert.)

Man beachte, daß die Hilfsfunktion ϑ, ebenso wie die Zustandsübergangsfunktion τ, einerseits mit dem gewöhnlichen Zustand $r \in \Theta$ und andererseits mit erweiterten Zuständen $t \in T' \subseteq \Theta \times \Omega$ aufgerufen wird.

Der initiale Zustand wird analog zu einem Konjunktions-Quantorzustand definiert:

$$\sigma(x) = [\circ, y, p, \sigma(y), \varnothing].$$

4.5.7.4 Parallele Quantorausdrücke

Multimengen

Eine *Multimenge* M ist, anschaulich gesprochen, eine Menge, die Elemente *mehrfach* enthalten kann. Formal kann M daher entweder als eine gewöhnliche Menge von *Paaren* $[e, n]$ definiert werden, wobei e ein *Element* der Multimenge und $n \in \mathbb{N}_0$ seine *Kardinalität* darstellt, oder aber als eine *Funktion* $M: E \to \mathbb{N}_0$, die jedem Element e (aus einer vorgegebenen Grundmenge E) seine Kardinalität $n = M(e)$ in M zuordnet.

Sind M_1 und M_2 zwei Multimengen, so bezeichnet $M = M_1 + M_2$ ihre Vereinigung oder *Summe*, die als Summe der Funktionen M_1 und M_2 definiert ist. Die Multimenge M enthält ein Element e also genau (n_1+n_2)-mal, wenn es in M_1 n_1-mal und in M_2 n_2-mal enthalten ist.

Analog ist auch die *Differenz* $M_1 - M_2$ definiert, sofern durch die Subtraktion keine negativen Kardinalitäten entstehen.

Aus Gründen der Bequemlichkeit wird im folgenden auch die Summe bzw. Differenz einer Multimenge M und eines einzelnen Elements e als $M \pm e = M \pm \chi_e$ definiert.[16] Insbesondere bezeichnet $M' = M - e + e'$ diejenige Multimenge, die man erhält, wenn man in der Multimenge M ein Vorkommen des Elements e durch das Element e' ersetzt.

Außerdem wird die Summe mehrerer Elemente $e_1, \ldots, e_n$ (sofern es sich um Elemente wie z. B. Tupel handelt, für die anderweitig keine Addition definiert ist) als Multimenge

$$M = \sum_{i=1}^{n} e_i = \sum_{i=1}^{n} \chi_{e_i} = \chi_{e_1} + \ldots + \chi_{e_n}$$

definiert, die genau diese Elemente enthält.

Schließlich wird auch die spezielle Kardinalität ∞ zugelassen, für die die Beziehung $\infty \pm n = \infty$ für $n \in \mathbb{N}_0$ gelten soll. Mit Hilfe der Summenschreibweise $M = \sum_{i=1}^{\infty} e$ läßt sich beispielsweise eine Multimenge M beschreiben, die das Element e „unendlich oft" enthält.

Das Prädikat $e \in M$ sei genau dann wahr, wenn das Element e in der Multimenge M mit positiver Kardinalität vorkommt, d. h. wenn $M(e) > 0$ gilt.

Die Teilmengenbeziehung $M_1 \subseteq M_2$ sei erfüllt, wenn jedes Element $e \in M_1$ in der Multimenge M_2 mit mindestens derselben Kardinalität wie in M_1 enthalten ist, d. h. wenn für die Funktionen M_1 und M_2 die Beziehung $M_1 \leq M_2$ gilt.

Zwei Multimengen M_1 und M_2 sind gleich, wenn sie dieselben Elemente jeweils mit derselben Kardinalität enthalten, d. h. wenn M_1 und M_2 als Funktionen gleich sind. Da man auch eine gewöhnliche Menge als Multimenge interpretieren kann, die jedes ihrer Elemente genau einmal enthält, ist auf diese Weise auch die Gleichheit zwischen einer Multimenge M_1 und einer gewöhnlichen Menge M_2 definiert.

Definitionen

Ein *paralleler Quantorzustand*, d. h. ein Zustand eines parallelen Quantorausdrucks $x \equiv \underset{p}{\circledcirc}\, y$, ist ein 4-Tupel $s = [\circledcirc, y, p, A]$ mit einer Menge A von Multimengen, die als *Alternativen* des Zustands s bezeichnet werden. Jede Alternative $T \in A$ besteht aus erweiterten Zuständen $[t, \omega]$ mit einem Wert $\omega \in \Omega \cup \{ p \}$ (vgl. § 4.5.6.4) und einem Zustand t des zugehörigen Zweigs y_p^{ω} bzw. $y_p^p \equiv y$. Die Gesamtkardinalität einer Alternative ist unendlich, die Anzahl der *verschiedenen* Elemente ist jedoch

[16] $\chi_e = \chi_{\{e\}}$ bezeichne die *charakteristische Funktion* der Menge $\{ e \}$, für die gilt: $\chi_e(e) = 1$, $\chi_e(e') = 0$ für $e' \neq e$. Somit entspricht χ_e einer Multimenge, die das Element e genau einmal enthält.

endlich. Außerdem enthält jede Alternative genau *einen* erweiterten Zustand (nämlich das Paar $[\sigma(y),\ p]$) mit unendlicher Kardinalität.

Neben dieser Multimenge von Alternativen enthält ein paralleler Quantorzustand − ebenso wie alle anderen Quantorzustände − den Rumpf y sowie den Parameter p des Ausdrucks x, die beide für die Definition des Zustandsübergangs benötigt werden.

Anmerkung: Auf den ersten Blick mag die Verwendung von Multimengen-Elementen mit unendlicher Kardinalität unnötig kompliziert erscheinen. Außerdem könnte man auf das Zustand/Wert-Paar $[\sigma(y),\ p]$ in einer Alternative vollständig verzichten, da sowohl y als auch p direkt im Zustand s enthalten sind. Es zeigt sich jedoch, daß die Integration dieses Paars in jede Alternative zu einer homogeneren und einfacheren Definition des Zustandsübergangs führt, da man so den initialen Zustand $\sigma(y)$ wie jeden anderen Zustand des abstrakten Zweigs y behandeln kann. Die Rechenregel $\infty - n = \infty$ für die spezielle Kardinalität ∞ sorgt hierbei automatisch dafür, daß das Paar $[\sigma(y),\ p]$ in jeder Alternative erhalten bleibt, auch wenn es ein- oder mehrmals durch konkretisierte Paare ersetzt wird (siehe unten).

Anders als bei den übrigen Quantorzuständen, trägt auch die homogene Behandlung von Zuständen des abstrakten und der konkreten Zweige mit Hilfe des Pseudowerts p (vgl. § 4.5.6.4) und ihre Zusammenfassung in jeweils *einer* Multimenge pro Alternative zu einer Vereinfachung der nachfolgenden Definitionen bei.

Ein paralleler Quantorzustand $s = [\circledcirc,\ y,\ p,\ A]$ ist genau dann ein *gültiger Zustand* bzw. ein *Endzustand*, wenn er mindestens eine Alternative besitzt, die nur gültige bzw. Endzustände enthält:

$$\psi(s) = \bigvee_{T \in A} \bigwedge_{t \in T} \psi(t), \qquad \varphi(s) = \bigvee_{T \in A} \bigwedge_{t \in T} \varphi(t).$$

Beim *Zustandsübergang* wird − ähnlich wie bei der elementaren parallelen Komposition − aus jeder Alternative S eine Menge neuer Alternativen $S - t + t'$ erzeugt, bei denen jeweils ein erweiterter Zustand $t \in S$ durch den transformierten Zustand $t' = \tau_a(t)$ ersetzt wird. Bei den Alternativen S handelt es sich jedoch nicht um die ursprünglichen Alternativen $T \in A$, sondern um modifizierte Multimengen $S = T - \sum_{i=1}^{k} [t_i,\ p] + \sum_{i=1}^{k} \left[(t_i)_p^{\omega_i},\ \omega_i\right]$, bei denen jeweils k erweiterte Zustände $[t_i,\ p] \in T$ des abstrakten Zweigs y durch konkretisierte Zustand/Wert-Paare $\left[(t_i)_p^{\omega_i},\ \omega_i\right]$ ersetzt wurden. Hierbei entsprechen die Werte $\omega_1, \ldots, \omega_k$ genau denjenigen relevanten Parameterwerten der Aktion a, die in der ursprünglichen Alternative T *nicht* vorkommen:

$$\tau_a(s) = [\circledcirc,\ y,\ p,\ A'] \quad \text{mit}$$

$$A' = \{\, S - t + \tau_a(t) \mid t \in S,\ S \in \vartheta(T),\ T \in A \,\} \quad \text{und}$$

$$\vartheta(T) = \left\{ T - \sum_{i=1}^{k} [t_i,\ p] + \sum_{i=1}^{k} \left[(t_i)_p^{\omega_i},\ \omega_i\right] \;\middle|\; \sum_{i=1}^{k} [t_i,\ p] \subseteq T,\ \sum_{i=1}^{k} \omega_i = \Omega_a(y,\ p) \setminus \Omega(T) \right\}.$$

Ein paralleler Quantorzustand kann − ähnlich wie ein paralleler Kompositionszustand − *optimiert* werden, indem Alternativen mit ungültigen Teilzuständen eliminiert werden:

$$\rho_1(s) = [\circledcirc,\ y,\ p,\ \hat{A}] \quad \text{mit} \quad \hat{A} = \left\{ T \in A \;\middle|\; \bigwedge_{t \in T} \psi(t) \right\}.$$

Der *initiale Zustand* eines parallelen Quantorausdrucks $x \equiv \underset{p}{\circledcirc}\, y$ enthält genau eine Alternative, die den erweiterten Zustand $[\sigma(y),\ p]$ unendlich oft enthält:

$$\sigma(x) = [\circledcirc,\ y,\ p,\ A] \quad \text{mit} \quad A = \left\{ \sum_{i=1}^{\infty} [\sigma(y),\ p] \right\}.$$

4.6 Implementierung des Zustandsmodells

4.6.1 Vorbereitungen

Die in den Abschnitten 4.3.3.1 und 4.4.2.4 vorgestellten Konzepte der *offenen Typen* und *partiellen Funktionen* erlauben es, die mathematischen Definitionen der Abschnitte 4.5.4 bis 4.5.7 relativ direkt in CH-Programmcode umzusetzen. Da die Korrektheit des operationalen Modells in Anhang B bewiesen wird, erhält man so auch eine *weitgehend verifizierte Implementierung* von Interaktionsausdrücken, obwohl die Übertragung der mathematischen Funktionsdefinitionen in CH-Funktionen nur „intuitiv" und nicht streng formal verifiziert ist.

Aus Platzgründen werden im folgenden nur einige ausgewählte Kategorien von Ausdrücken betrachtet, da sich die Definitionen der übrigen Kategorien in ähnlicher Weise umsetzen lassen.

4.6.1.1 Zustände

Der Zustand eines Ausdrucks wird als offener Typ `State` vereinbart, der zunächst lediglich ein Attribut `expr` vom Typ `Expr` besitzt (vgl. § 4.3.1), das auf den zugehörigen Ausdruck verweist:

```
// Zustand.
type State;
attr expr: State -> Expr;       // Zugehöriger Ausdruck.
```

Dieses Attribut, das so im mathematischen Modell *nicht* existiert, erlaubt es, mit Hilfe *abgeleiteter Attribute* implizit auf gewisse Zustandskomponenten zuzugreifen, die im mathematischen Modell explizit repräsentiert werden:

```
attr cat: State -> Category     // Kategorie des Zustands.
    = expr/cat;

attr action: State -> Action    // Aktion eines atomaren Zustands.
    = expr/action;

attr param: State -> Param      // Parameter eines Quantorzustands.
    = expr/param;
```

Aufgrund dieser Deklarationen ist die Formulierung `s/action` für ein Objekt `s` vom Typ `State` beispielsweise äquivalent zu `s/expr/action` und liefert daher den Wert des Attributs `action` des `Expr`-Objekts `s/expr`.

Die beiden Sonderzustände $\bot$ und $\top$ werden durch die Konstanten `nil` (vgl. § 4.3.3.2) und `Success` repräsentiert, wobei letztere als spezielle Ausprägung des Typs `State` vereinbart wird:[17]

```
inst State { Success };
```

4.6.1.2 Funktionen

Die *Initialisierungsfunktion* σ wird durch eine stückweise definierte CH-Funktion `init()` implementiert, deren Teile jeweils Ausdrücke `x` einer bestimmten Kategorie `x/cat` behandeln:

```
// Initialer Zustand eines Ausdrucks x der Kategorie ...
part State init(Expr x) if (x/cat == ...) {
    ...
}
```

[17] Wie man sieht, schließen sich `attr`- und `inst`-Deklarationen für ein und denselben Typ nicht gegenseitig aus. Mit `inst` vereinbarte Konstanten wie z. B. `Success` unterscheiden sich per Definition von allen übrigen Objekten ihres Typs.

In ähnlicher Weise wird das Zustandsprädikat φ, das darüber Auskunft gibt, ob ein Zustand ein *Endzustand* ist, durch eine partielle Funktion `final()` repräsentiert, die die Zustandskategorie `s/cat` (d. h. eigentlich `s/expr/cat`) als Auswahlkriterium verwendet:

```
// Endzustands-Indikator eines Zustands s der Kategorie ...
part bool final(State s) if (s/cat == ...) {
    ...
}
```

Die *optimierte Zustandsübergangsfunktion* $\hat{\tau}$ schließlich, die eine Kombination der einfachen Zustandsübergangsfunktion τ und der Optimierungsfunktion ρ darstellt, wird durch eine partielle Funktion `trans()` repräsentiert, die ebenfalls anhand der Zustandskategorie `s/cat` „verzweigt“:

```
// Optimierter Zustandsübergang für einen Zustand s der Kategorie ...
part State trans(State s, Action a) if (s/cat == ...) {
    ...
}
```

Auf eine explizite Implementierung des Prädikats ψ, das anzeigt, ob ein Zustand *gültig* ist, kann in einem optimierten Zustandsmodell verzichtet werden, da ungültige Zustände aufgrund der Optimierung

$$\rho_3(s) = \bot, \quad \text{falls } \psi(s) = \bot \quad \text{(vgl. § 4.5.3.2)}$$

stets durch den äquivalenten Zustand $\bot$ bzw. `nil` ersetzt werden.

4.6.2 Implementierung elementarer Ausdrücke

4.6.2.1 Atomare Ausdrücke

Abbildung 4.14 zeigt die implementierungstechnische Umsetzung der mathematischen Definitionen aus § 4.5.4.1 für atomare Ausdrücke. Um die hierbei auftretenden Sonderzustände `Success` und `nil` korrekt zu behandeln, weichen die Definitionen der partiellen Funktionen `final()` und `trans()` teilweise von dem in § 4.6.1.2 skizzierten Schema ab.

4.6.2.2 Boolesche Operatoren

Zusätzliche Zustandsattribute

Um die Definitionen der Abschnitte 4.5.4.2 bis 4.5.4.4 (Disjunktion, Option und Konjunktion) implementierungstechnisch umsetzen zu können, wird der Typ `State` zunächst um zwei Attribute `left` und `right` (jeweils vom Typ `State`) erweitert, die den Teilzuständen l und r eines disjunktiven bzw. konjunktiven Zustands $s = [\bigcirc, l, r]$ bzw. $s = [\bullet, l, r]$ entsprechen:

```
// Zusätzliche Zustandsattribute.
attr left: State -> State;      // Zustände des linken und
attr right: State -> State:     // rechten Teilausdrucks.
```

Rein *syntaktisch* haben diese Attribute nichts mit den gleichnamigen Attributen des Typs `Expr` zu tun (ebenso wie Komponenten verschiedener Strukturen in C vollkommen unabhängig voneinander sind), *semantisch* besteht jedoch ein Zusammenhang: Das Attribut `s/left` (bzw. `s/right`) eines Zustands `s` verweist jeweils auf einen Zustand des Teilausdrucks `s/expr/left` (bzw. `s/expr/right`) des zu `s` gehörenden Ausdrucks `s/expr`.

```
// Initialer Zustand eines atomaren Ausdrucks.
part State init(Expr x) if (x/cat == Atom) {
    return State(expr = x);
}

// Endzustands−Indikator eines atomaren Zustands.
part bool final(State s) if (s/cat == Atom || s == nil) {
    return false;
}
part bool final(State s) if (s == Success) {
    return true;
}

// Optimierter Zustandsübergang für einen atomaren Zustand.
part State trans(State s, Action a) if (s/cat == Atom) {
    if (a == s/action) return Success;
    else return nil;
}
part State trans(State s, Action a) if (s == Success || s == nil) {
    return nil;
}
```

Abbildung 4.14: Implementierung atomarer Ausdrücke

Disjunktion

Nach diesen Vorbereitungen kann die Disjunktion gemäß Abb. 4.15 implementiert werden. Analog zu den mathematischen Definitionen, rufen sich auch die `CH`-Funktionen `init()`, `final()` und `trans()` jeweils rekursiv auf, um die Teilausdrücke `x/left` und `x/right` des Ausdrucks `x` bzw. die Teilzustände `s/left` und `s/right` des Zustands `s` zu behandeln. Der Mechanismus der partiellen

```
// Initialer Zustand einer Disjunktion.
part State init(Expr x) if (x/cat == Disj) {
    return State(expr = x, left = init(x/left), right = init(x/right));
}

// Endzustands−Indikator eines disjunktiven Zustands.
part bool final(State s) if (s/cat == Disj) {
    return final(s/left) || final(s/right);
}

// Optimierter Zustandsübergang für einen disjunktiven Zustand.
part State trans(State s, Action a) if (s/cat == Disj) {
    State l = trans(s/left, a);
    State r = trans(s/right, a);

    if (!l) return r;
    if (!r) return l;
    return State(expr = s/expr, left = l, right = r);
}
```

Abbildung 4.15: Implementierung der Disjunktion

Funktionen sorgt hierbei automatisch dafür, daß bei jedem Funktionsaufruf der passende Zweig der jeweiligen Funktion ausgewählt wird, obwohl viele dieser Zweige erst im weiteren Verlauf des Programms definiert werden.

Beim Zustandsübergang `trans()` werden die durch die Funktion ρ_2 definierten Optimierungen vorgenommen, wenn einer der transformierten Teilzustände *t* (`l` oder `r`) ungültig ist, was gemäß § 4.6.1.2 äquivalent zu der Bedingung (*t*`==nil`) bzw. (`!`*t*) ist. Wenn keiner dieser Spezialfälle vorliegt, wird ein neuer Zustand erzeugt, dessen `expr`-Attribut vom ursprünglichen Zustand `s` übernommen wird.

Option

Eine Option $\curlyvee y$ wird gemäß § 4.5.4.3 durch eine spezielle Disjunktion $y \bigcirc \varepsilon$ mit $\sigma(\varepsilon) = \top$ ersetzt. Da der leere bzw. fehlende Ausdruck ε in natürlicher Weise durch die Konstante `nil` repräsentiert wird, muß der initiale Zustand dieses speziellen Ausdrucks durch einen entsprechenden Zweig der Funktion `init()` definiert werden:

```
// Initialer Zustand eines leeren Ausdrucks.
part State init(Expr x) if (x == nil) {
    return Success;
}
```

Konjunktion und Synchronisation

Die Konjunktion kann vollkommen analog zur Disjunktion implementiert werden.

Bei einer Synchronisation $x \equiv y \mathbin{\mathbf{o}} z$ ist zu beachten, daß die Teilausdrücke y und z eines Zustands $s = [\mathbf{o}, y, z, l, r]$ nicht explizit repräsentiert werden müssen, da sie indirekt über die Attribute `s/expr/left` und `s/expr/right` des Zustands `s` zugänglich sind. Der für den Zustandsübergang erforderliche „Alphabettest" $a \in \alpha(z) \setminus \alpha(y)$ bzw. $a \in \alpha(y) \setminus \alpha(z)$ (vgl. § 4.5.4.5) kann mit Hilfe einer Funktion

```
bool contains(Expr x, Action a);
```

realisiert werden, die rekursiv überprüft, ob die Aktion `a` im Alphabet des Ausdrucks `x` enthalten ist oder nicht.[18] Unter Verwendung dieser Hilfsfunktion kann auch die Synchronisation ähnlich zur Disjunktion und Konjunktion implementiert werden.

4.6.2.3 Sequentielle Operatoren

Sequentielle Komposition

Zur Repräsentation sequentieller Kompositionszustände $s = [-, z, l, R]$ wird der Typ `State` erneut erweitert, diesmal um ein mehrwertiges Attribut `rights`[19] zur Speicherung der Zustandsmenge R (vgl. Abb. 4.16). Auf eine explizite Repräsentation des Teilausdrucks z kann − ähnlich wie der Implementierung der Synchronisation in § 4.6.2.2 − verzichtet werden, da er indirekt über das Attribut `s/expr/right` des Zustands `s` zugänglich ist.

Gemäß § 4.5.4.6 enthält die Menge R beim initialen Zustand einer sequentiellen Komposition genau dann den initialen Zustand $\sigma(z)$ des rechten Teilausdrucks, wenn der initiale Zustand `l` des linken Teilausdrucks ein Endzustand ist. Daher wird die Variable `R` in der Funktion `init()` zunächst mit der

---

[18] Da die Wertemenge Ω einerseits unendlich groß und andererseits nicht konkret bekannt ist, läßt sich die formale Definition $\alpha(x) = \bigcup_{\omega \in \Omega} \alpha\left(y_p^\omega\right)$ des Alphabets eines Quantorausdrucks $x \equiv \underset{p}{\bigcirc}\, y$ allerdings nicht direkt implementierungstechnisch umsetzen. Stattdessen müssen Quantorparameter, denen momentan kein bestimmter Wert zugeordnet ist, beim Vergleich der Aktion `a` mit einer Aktion `b` des Ausdrucks `x` als *Platzhalter* für einen beliebigen Wert behandelt werden. Sofern derselbe Parameter in einer Aktion `b` mehrmals als Argument auftritt, muß er allerdings an allen Stellen *denselben* Wert repräsentieren.

[19] Per Konvention enden die Namen mehrwertiger Attribute jeweils mit einem Plural-s.

```
// Zusätzliches Zustandsattribut.
attr rights: State ->> State;   // Zustände des rechten Teilausdrucks.

// Initialer Zustand einer sequentiellen Komposition.
part State init(Expr x) if (x/cat == Sequ) {
    State l = init(x/left);
    Set(State) R = set(); if (final(l)) R |= init(x/right);
    return State(expr = x, left = l, rights = R);
}

// Endzustands-Indikator eines sequentiellen Kompositions-Zustands.
part bool final(State s) if (s/cat == Sequ) {
    State r;
    forall (r << s/rights) if (final(r)) return true;
    return false;
}

// Optimierter Zustandsübergang für einen sequ. Kompositions-Zustand.
part State trans(State s, Action a) if (s/cat == Sequ) {
    State l = trans(s/left, a);
    Set(State) R = set(); if (final(l)) R |= init(s/expr/right);

    State r, r_;
    forall (r << s/rights) if (r_ = trans(r, a)) R |= r_;

    if (*R == 1) if (!l || l == Success) return R[1];

    if (*R || l) return State(expr = s/expr, left = l, rights = R);
    else return nil;
}
```

Abbildung 4.16: Implementierung der sequentiellen Komposition

leeren Menge `set()` initialisiert und anschließend (mit Hilfe des Operators `|=`) ggf. um das Element `init(x/right)` erweitert.[20]

In der Funktion `final()` wird mit Hilfe der Iterationsanweisung `forall` überprüft, ob die Menge `s/rights` einen Teilzustand `r` enthält, der ein Endzustand ist.[21] Wenn dies der Fall ist, liefert die Funktion `true`, andernfalls `false`.

Beim Zustandsübergang `trans()` wird der Teilzustand `s/left` rekursiv transformiert. Die Menge `R` wird in der gleichen Weise wie in der Funktion `init()` initialisiert und anschließend um die Menge aller *gültigen* Zustände `r_`[22] erweitert, die durch Transformation eines Zustands `r` ∈ `s/rights` entstehen. Durch die Beschränkung auf *gültige* Teilzustände `r_` wird bereits der Teil ρ_1 der Optimierungsfunktion ρ implementiert, während die verbleibenden Teile ρ_2 und ρ_3 durch die Fallunterscheidung am Ende der Funktion `trans()` abgedeckt werden.

---

[20] Da *Mengen* in `CH` als sortierte, duplikatfreie *Sequenzen* repräsentiert werden (vgl. § A.3.2.2), ist der Typ `Set(State)` syntaktisch äquivalent zu `Seq(State)` (vgl. § 4.4.2.1). Neben typischen Listen- und Array-orientierten Operationen (wie z. B. Einfügen am Anfang und Ende und Elementzugriff mittels Index) unterstützen Sequenzen daher auch mengenbezogene Operationen wie Vereinigung, Durchschnitt und Elementtest.
Aufgrund ihrer Verwandtschaft zur logischen Oder-Operation wird die Vereinigung zweier Mengen durch den Oder-Operator `|` notiert. Dem Stil von C bzw. C++ folgend, wird daher durch den Operator `|=` die Kombination von Vereinigung und Zuweisung ausgedrückt. Zur weiteren Vereinfachung der Notation akzeptiert dieser Operator auf der rechten Seite nicht nur eine Menge, sondern auch ein einzelnes Element, das in die Menge auf der linken Seite eingefügt wird. Die Anweisung `R |= r` ist daher äquivalent zu `R = R | set(r)`, kann aber − abgesehen von der kompakteren Notation − effizienter implementiert werden.

[21] Der Operator << steht quasi für den Elementoperator ∈. Die Anweisung `forall(r << s/rights)` kann daher gelesen werden als: ∀ `r` ∈ `s/rights`.

[22] Gesprochen „r Strich“, so wie r'.

Sequentielle Iteration

Da das Zustandsmodell der sequentiellen Iteration eng mit dem der sequentiellen Komposition verwandt ist, kann es auf ähnliche Art und Weise implementierungstechnisch umgesetzt werden.

4.6.2.4 Parallele Operatoren

Da in § 4.6.4.2 die implementierungstechnische Umsetzung des Zustandsmodells paralleler *Quantoren* erläutert wird, wird an dieser Stelle auf die konzeptuell einfachere Implementierung der *binären* parallelen Komposition nicht näher eingegangen.

Auf eine direkte Implementierung der parallelen Iteration kann, wie bereits erwähnt, verzichtet werden, da sie bei der syntaktischen Analyse eines Ausdrucks durch einen äquivalenten Quantorausdruck ersetzt wird (vgl. § 4.3.1).

4.6.3 Vorbereitungen für Quantorausdrücke

4.6.3.1 Konkretisierung von Aktionen, Ausdrücken und Zuständen

Im Gegensatz zu § 3.3.3.1 und § 4.5.6.1, wo durch die Konkretisierung einer Aktion, eines Ausdrucks oder eines Zustands s ein *neues* (im wesentlichen zwar dupliziertes, stellenweise aber modifiziertes) mathematisches Objekt s_p^ω entsteht, werden abstrakte Objekte im folgenden durch eine geeignete *An-Ort-und-Stelle-Modifikation* (engl. in place modification) *temporär* in konkretisierte Objekte „verwandelt“. Auf diese Weise kann ein abstraktes Objekt bei Bedarf ohne aufwendige Kopieroperationen konkretisiert werden, und ggf. kann ein und dasselbe abstrakte Objekt zu verschiedenen Zeitpunkten mehrere verschiedene konkretisierte Objekte repräsentieren.

Konkret wird hierfür − mit Hilfe der `CH`-Anweisung `p(value = v)`[23] − das Attribut `value` eines Parameter-Objekts `p` (das für einen Quantorparameter bisher unbenutzt war) mit einem bestimmten Wert `v` belegt und so zum Ausdruck gebracht, daß der Parameter `p` momentan diesen Wert `v` repräsentiert. Da ein Parameter-Objekt gemäß § 4.3.2 von sämtlichen Aktions-Objekten, die diesen Parameter in ihrer Argumentliste enthalten, als gemeinsames Objekt referenziert wird, werden durch diese Wertzuweisung *alle* betroffenen Aktionen (und damit auch alle Ausdrücke und Zustände, die entsprechende Aktionen als Teilobjekte enthalten) *auf einmal* konkretisiert.

Mit Hilfe der Anweisung `p(~value)` kann der Wert des Attributs `p/value` wieder entfernt werden, wodurch die Konkretisierung sozusagen rückgängig gemacht wird − eine Operation, die so im mathematischen Modell nicht möglich ist, weil durch die Konkretisierung eines Objekts die Information verlorengeht, an welchen Stellen sich der Quantorparameter `p` zuvor befand.

4.6.3.2 Relevante Parameterwerte

Ähnlich wie die in § 4.6.2.2 erwähnte Funktion `contains()`, die überprüft, ob eine Aktion a in einem Ausdruck x enthalten ist, läßt sich eine `CH`-Funktion

```
Set(Value) relvals(Expr x, Action a);
```

implementieren, die die Menge $\Omega_a(y, p)$ der relevanten Parameterwerte einer Aktion a bezüglich des Ausdrucks $y \mathrel{\hat=}$ `x/body` und des Parameters $p \mathrel{\hat=}$ `x/param` gemäß der Definition in § 4.5.6.2 bestimmt. Hierfür wird der Ausdruck y rekursiv durchlaufen und für jeden atomaren Teilausdruck b überprüft, ob die Bedingungen $a \neq b$ und $a = b_p^\omega$ für geeignete Werte ω erfüllt sind, d. h. ob die Aktion b durch die Konkretisierung b_p^ω gleich a wird. Aufgrund der Inklusion $\Omega_a(y, p) \subseteq \{ a_1, \ldots, a_n \}$

[23] Die Syntax der Anweisung `p(value = v)` entspricht der eines Konstruktoraufrufs `Param(value = v)` (vgl. § 4.3.3.3); allerdings steht vor der Klammer kein Typname, sondern eine Variable (oder ein entsprechender Ausdruck), die ein Objekt dieses Typs beinhaltet.

für $a = [a_0, a_1, \ldots, a_n]$ müssen für ω lediglich die konkreten Parameterwerte der Aktion a eingesetzt werden.[24]

4.6.3.3 Erweiterte Zustände

Analog zu § 4.5.6.4 wird in Abb. 4.17 ein Typ `Pair` zur Repräsentation von Zustand/Wert-Paaren vereinbart und die Definition der Zustandsübergangsfunktion `trans()` auf derartige erweiterte Zustände ausgedehnt.[25] Neben den Argumenten `s` und `a` erhält diese Variante der Funktion `trans()` einen Quantorparameter `p`, über den sie − wie in § 4.6.3.1 beschrieben − den Zustand `s/state` vor Durchführung des Zustandsübergangs konkretisieren kann. Aus Gründen, deren genaue Erläuterung hier zu weit führen würde, muß diese Konkretisierung anschließend gleich wieder rückgängig gemacht werden, um eine unbeabsichtigte Interpretation des Parameters `p` als Wert `s/value` an anderen Stellen des Programms zu vermeiden.

```
// Zustand/Wert-Paar.
type Pair;
attr state: Pair -> State;      // Zustand.
attr value: Pair -> Value;      // Wert.

// Optimierter Zustandsübergang für ein Zustand/Wert-Paar.
Pair trans(Pair s, Action a, Param p) {
    p(value = s/value);
    State t = trans(s/state, a);
    p(˜value);

    if (t) return Pair(state = t, value = s/value);
    else return nil;
}
```

Abbildung 4.17: Zustand/Wert-Paare

4.6.4 Implementierung von Quantorausdrücken

4.6.4.1 Boolesche Quantorausdrücke

Disjunktion

Abbildung 4.18 zeigt die Implementierung des Zustandsmodells für Disjunktions-Quantorausdrücke, wie es in § 4.5.7.1 definiert wurde.

Da eine binäre Disjunktion und ein Disjunktions-Quantorausdruck gemäß Tab. 4.6 (§ 4.3.1) beide zur Kategorie `Disj` gehören und daher anhand des Kriteriums `(x/cat == Disj)` nicht unterschieden werden können, wird hier zusätzlich die Existenz des Attributs `x/param` überprüft (das nur bei Quantorausdrücken vorhanden ist), um einen Quantorausdruck von einem binären Ausdruck derselben Kategorie unterscheiden zu können. Hierbei wird ausgenutzt, daß beim Aufruf einer partiellen Funktion wie z. B. `init()` die Auswahlkriterien der einzelnen Zweige in *umgekehrter Definitionsreihenfolge* ausgewertet werden und jeweils der *erste* Zweig, dessen Bedingung erfüllt ist, ausgeführt wird. Dar-

[24] Tatsächlich genügt es sogar, diejenigen Werte a_i zu betrachten, für die $b_i = p$ gilt. Außerdem müssen nur Aktionen b überprüft werden, die dieselbe *Signatur* (d. h. denselben Namen und dieselbe Anzahl von Argumenten) wie a besitzen.

[25] Da die hier definierte Funktion `trans()` eine andere Signatur besitzt als die bisher betrachtete, stückweise definierte Funktion `trans()`, handelt es sich, syntaktisch gesehen, um zwei vollkommen verschiedene Funktionen, die nur „zufällig“ denselben Namen besitzen. Die Funktion `trans()` wird also sowohl statisch als auch dynamisch überladen. Wenn man diese doppelte Überladung eines Funktionsnamens als verwirrend empfindet, könnte man die hier definierte Funktion auch anders benennen.

```
// Zusätzliche Zustandsattribute.
attr state: State -> State;     // Zustand des abstrakten Zweigs.
attr pairs: State ->> Pair;     // Zustand/Wert-Paare der
                                // konkretisierten Zweige.

// Initialer Zustand eines Disjunktions-Quantorausdrucks.
part State init(Expr x) if (x/cat == Disj && x/param) {
    return State(expr = x, state = init(x/body), pairs = set());
}

// Endzustands-Indikator eines disjunktiven Quantorzustands.
part bool final(State s) if (s/cat == Disj && s/param) {
    Pair t;
    forall (t << s/pairs) if (final(t/state)) return true;
    return final(s/state);
}

// Optimierter Zustandsübergang für einen disjunktiven Quantorzustand.
part State trans(State s, Action a) if (s/cat == Disj && s/param) {
    Param p = s/param;
    State r = trans(s/state, a);
    Set(Pair) T = set();
    Set(Value) V = relvals(s/expr, a);

    Pair t, t_;
    forall (t << s/pairs) {
        if (t_ = trans(t, a, p)) T |= t_;
        else if (r) T |= Pair(state = nil, value = t/value);
        V -= t/value;
    }

    Value v;
    forall (v << V) {
        Pair t(state = s/state, value = v);
        if (t_ = trans(t, a, p)) T |= t_;
        else if (r) T |= Pair(state = nil, value = v);
    }

    if (!r && *T == 1 && T[1]/state == Success) return Success;
    if (r || *T) return State(expr = s/expr, state = r, pairs = T);
    return nil;
}
```

Abbildung 4.18: Implementierung von Disjunktions-Quantorausdrücken

aus folgt − unter der Annahme, daß die Zweige in derselben Reihenfolge implementiert sind, wie sie im Text vorgestellt werden −, daß die Bedingung (`x/cat == Disj && x/param`) aus Abb. 4.18 *vor* der Bedingung (`x/cat == Disj`) aus Abb. 4.15 (§ 4.6.2.2) überprüft wird und somit für einen Disjunktions-*Quantorausdruck* (für den die erstgenannte Bedingung erfüllt ist) der Zweig in Abb. 4.18 ausgeführt wird, während für eine *binäre* Disjunktion (für die diese Bedingung nicht erfüllt ist) der Zweig in Abb. 4.15 gewählt wird. Auf diese Weise ist es − ähnlich wie in objektorientierten Programmiersprachen − möglich, eine allgemeine Funktion oder Methode, die auf einer bestimmten Menge

von Objekten definiert ist, nachträglich durch eine *speziellere* Variante zu überschreiben, die auf einer *Teilmenge* dieser Objekte operiert.[26]

Beim Zustandsübergang wird in der ersten `forall`-Schleife die Menge aller transformierten Zustand/Wert-Paare `t_` für `t` ∈ `s/pairs` bestimmt und gleichzeitig die zugehörigen Werte `t/value` (mit Hilfe des Operators `-=`) aus der Menge `V` der relevanten Parameterwerte der Aktion `a` entfernt, so daß die verbleibende Menge `V` der Menge $\Omega_a(y, p) \setminus \Omega(T)$ entspricht. Die Werte dieser Menge werden in der zweiten `forall`-Schleife dazu verwendet, den Zustand `s/state` des abstrakten Zweigs zu konkretisieren und dann ebenfalls zu transformieren. In beiden Schleifen wird jeweils der Teil ρ_1 der Optimierungsfunktion ρ mitimplementiert, indem Zustand/Wert-Paare mit ungültigem Zustand nur dann zur Menge `T` hinzugefügt werden, wenn der Zustand `r` des abstrakten Zweigs gültig ist. Die abschließende Fallunterscheidung implementiert die übrigen Optimierungen ρ_2 und ρ_3.

Konjunktion und Synchronisation

Die beiden anderen „Booleschen Quantoren" (Konjunktion und Synchronisation) werden sehr ähnlich implementiert, wobei für den Zustandsübergang der Synchronisation wieder die in § 4.6.2.2 erwähnte Funktion `contains()` benötigt wird.

4.6.4.2 Parallele Quantorausdrücke

Multimengen

Analog zu gewöhnlichen Mengen, könnte man prinzipiell auch Multimengen mit Hilfe sortierter − aber nicht notwendigerweise duplikatfreier − Sequenzen repräsentieren. Allerdings besitzt dieser Ansatz gewisse Nachteile:

1. Je nach Kardinalität der einzelnen Elemente, enthält eine Multimenge u. U. relativ viel Redundanz, d. h. es wird unnötig viel Speicherplatz verbraucht.
2. Bei einer Iteration über alle Elemente einer Multimenge wird der Rumpf der Iteration für mehrfach vorhandene Elemente entsprechend mehrfach durchlaufen, d. h. es wird unnötig Rechenzeit verbraucht.
3. Schließlich ist dieser Ansatz grundsätzlich nicht geeignet, um die spezielle Element-Kardinalität ∞ zu repräsentieren.

Aus diesen Gründen stellt die `CH`-Bibliothek einen speziellen Typ `Mset(T)` bereit, der eine Multimenge `M` mit dem Elementtyp `T` als *Sequenz von Paaren* `(e, n)` repräsentiert, wobei `e` ∈ `T` jeweils ein Element der Multimenge und `n` $\in \mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ seine Kardinalität darstellt. Typische Operationen auf solchen Multimengen sind `M += e` und `M -= e`, die *eine* Ausprägung des Elements `e` zur Multimenge `M` hinzufügen bzw. aus ihr entfernen, d. h. die zu `e` gehörende Kardinalität `n` entsprechend aktualisieren und das Paar `(e, n)` ggf. neu in die Sequenz aufnehmen oder aus ihr entfernen. Bei einer Iteration mittels `forall` wird jedes Element der Multimenge *genau einmal* durchlaufen, selbst wenn seine Kardinalität größer als eins ist. (Falls erforderlich, kann man jedoch explizit auf die Kardinalität eines Elements zugreifen. Vgl. auch § A.3.2.3.)

[26] Allerdings wäre man bei einer objektorientierten Vorgehensweise gezwungen, die Funktion `init()` als Methode der Klasse `Expr` zu vereinbaren (die dann Unterklassen wie z. B. `QuantExpr`, `DisjQuantExpr` usw. besitzen würde), damit der übliche Polymorphie-Mechanismus greift. Da `init()` jedoch ein Objekt vom Typ `State` zurückliefert, erhält man auf diese Weise eine *syntaktische* Abhängigkeit der Klasse `Expr` von der Klasse `State`, die *konzeptionell* nicht existiert (Zustände basieren auf Ausdrücken und nicht umgekehrt!) und die Wiederverwendbarkeit von `Expr` in anderen Kontexten (in denen keine Zustände von Ausdrücken benötigt werden) einschränkt. Da ein Objekt vom Typ `State` darüber hinaus ein `Expr`-Objekt referenziert, erkauft man sich sogar eine *zyklische* Abhängigkeit zwischen `Expr` und `State`, die sich zumeist nur mit syntaktischen Kunstgriffen bewältigen läßt.
Da partielle Funktionen Laufzeit-Polymorphismus *unabhängig* von einer Klassenhierarchie unterstützen, treten derartige Probleme in `CH` nicht auf.

Initialer Zustand und Endzustands-Indikator

Abbildung 4.19 zeigt den ersten Teil der Implementierung paralleler Quantorausdrücke.

Der Typ `Alt` zur Repräsentation einer Alternative wird als Synonym bzw. Abkürzung des Typs `Mset(Pair)`, d. h. als Multimenge von Zustand/Wert-Paaren vereinbart. Das zusätzliche Zustandsattribut `alts` wird benötigt, da ein paralleler Quantorzustand gemäß § 4.5.7.4 im wesentlichen aus einer Menge von Alternativen besteht.

```
// Alternative.
type Alt = Mset(Pair);

// Zusätzliches Zustandsattribut.
attr alts: State ->> Alt;       // Menge von Alternativen.

// Initialer Zustand eines parallelen Quantorausdrucks.
part State init(Expr x) if (x/cat == Par && x/param) {
    Pair t(state = init(x/body), value = nil);
    Alt T = mset(t, oo);
    return State(expr = x, alts = set(T));
}

// Endzustands-Indikator eines parallelen Quantorzustands.
part bool final(State s) if (s/cat == Par && s/param) {
    Alt T; Pair t;
    forall (T << s/alts) {
        forall (t << T) { if (!final(t/state)) break; }
        else return true;
    }
    return false;
}
```

Abbildung 4.19: Implementierung paralleler Quantorausdrücke
(Teil 1: Initialer Zustand und Endzustands-Indikator)

In der Initialisierungsfunktion `init()` wird zunächst ein Zustand/Wert-Paar `t` konstruiert, das aus dem initialen Zustand des Quantorrumpfs `x/body` und dem Parameterwert `nil` besteht, der den „Pseudowert“ *p* (vgl. § 4.5.6.4) repräsentiert. Anschließend wird mit Hilfe der Multimengen-Konstruktorfunktion `mset()` eine Alternative `T` erzeugt, die dieses Paar `t` unendlich oft enthält,[27] und als Element des mehrwertigen Zustandsattributs `alts` vereinbart.

In der Funktion `final()` wird, zur möglichst effizienten Implementierung des Prädikats $\varphi(s) = \bigvee_{T \in A} \bigwedge_{t \in T} \varphi(t)$, eine Variante der `forall`-Anweisung verwendet, deren Idee aus der Programmiersprache Python [Lutz96, Lutz99] stammt: Eine Iterationsanweisung kann einen nachgestellten `else`-Teil besitzen, der genau dann ausgeführt wird, wenn die Schleife *vollständig* durchlaufen, d. h. *nicht* vorzeitig durch `break` beendet wurde. Die Anweisung `return true` wird daher ausgeführt, sobald die innere `forall`-Schleife einmal vollständig durchlaufen wurde, was genau dann der Fall ist, wenn die Alternative `T` nur Endzustände enthält. Wird die Schleife vorzeitig abgebrochen, weil einer der Teilzustände von `T` kein Endzustand ist, wird der `else`-Teil übersprungen und mit der nächsten Iteration der äußeren Schleife fortgefahren.

[27] Die Konstante `oo` (zwei kleine o’s) repräsentiert also die spezielle Kardinalität ∞.

Zustandsübergang

Beim Zustandsübergang (vgl. Abb. 4.20) wird mit Hilfe der Funktion `relvals()` zunächst, ebenso wie bei einem disjunktiven Quantorzustand, die Menge `V0` der relevanten Parameterwerte der Aktion `a` bestimmt. Anschließend wird für jede Alternative `T` ∈ `s/alts` eine Hilfsfunktion `transalt()` aufgerufen, die sämtliche neuen Alternativen bestimmt, die aus dieser Alternative „hervorgehen". Als Parameter erhält die Funktion hierfür:

1. die zu transformierende Alternative `T`;
2. die zu verarbeitende Aktion `a`;
3. den Quantorparameter `p`;
4. die Wertmenge `V`, die man erhält, wenn man aus einer Kopie `+V0`[28] der Menge `V0` alle Werte entfernt, die in der Alternative `T` bereits vorkommen;
5. (eine Referenz auf) die Menge `A`, in der die transformierten Alternativen gespeichert werden sollen.

Abschließend wird überprüft, ob der transformierte Gesamtzustand gültig ist, was genau dann der Fall ist, wenn die so erzeugte Menge `A` nicht leer ist.

```
// Optimierter Zustandsübergang für einen parallelen Quantorzustand.
part State trans(State s, Action a) if (s/cat == Par && s/param) {
    Param p = s/param;
    Set(Alt) A = set();
    Set(Value) V0 = relvals(s/expr, a);

    // Neue Menge von Alternativen bestimmen.
    Alt T; Pair t;
    forall (T << s/alts) {
        // Bereits "bekannte" Werte aus der Menge V0 entfernen.
        Set(Value) V = +V0;
        forall (t << T) V -= t/value;

        // Alternative T transformieren.
        // Neue Alternativen zur Menge A hinzufügen.
        transalt(T, a, p, V, A);
    }

    if (*A) return State(expr = s/expr, alts = A);
    else return nil;
}
```

Abbildung 4.20: Implementierung paralleler Quantorausdrücke (Teil 2: Zustandsübergang)

Transformation einzelner Alternativen

Die „erste Hälfte" der Prozedur[29] `transalt()` (vgl. Abb. 4.21) realisiert im Prinzip die in § 4.5.7.4 vorgenommene Abbildung einer Alternative T auf die Menge $\vartheta(T)$, indem sie (mit Hilfe rekursiver Aufrufe) in genau $k \mathrel{\hat{=}}$ `*V` Zustand/Wert-Paaren der Alternative T, die den Pseudowert $p \mathrel{\hat{=}}$ `nil` ent-

[28] Der Präfixoperator `+` erzeugt eine dynamische Kopie eines Objekts.
[29] Prozeduren, d. h. Funktionen ohne Rückgabewert, werden in `CH` durch das Schlüsselwort `proc` gekennzeichnet (vgl. § A.2.3).

```
// Transformation der Alternative T.
proc transalt(Alt T, Action a, Param p, Set(Value) V, Set(Alt) A,
                                                      int i = 1) {
    if (i <= *V) {
        Pair t;
        forall (t << T) if (t/value == nil) {
            Pair t_(state = t/state, value = V[i]);
            T -= t; T += t_;
            transalt(T, a, p, V, A, i+1);
            T += t; T -= t_;
        }
    }
    else {
        Pair t, t_;
        forall (t << T) if (t_ = trans(t, a, p)) {
            Alt T_ = +T; T_ -= t; T_ += t_;
            A |= T_;
        }
    }
}
```

Abbildung 4.21: Implementierung paralleler Quantorausdrücke
(Teil 3: Transformation von Alternativen)

halten, diesen Wert durch einen Wert $\omega_i \mathrel{\hat{=}}$ `V[i]` der Menge $\Omega_a(y, p) \setminus \Omega(T) \mathrel{\hat{=}}$ `V` ersetzt. Um aufwendige Kopieroperationen und unnötige temporäre Objekte zu vermeiden, werden die einzelnen Ersetzungen (mit Hilfe der oben erläuterten Multimengen-Operatoren `+=` und `-=`) an Ort und Stelle vorgenommen und später (durch Ausführen der „inversen" Operationen) wieder rückgängig gemacht.

Um die Menge `V` schrittweise durchlaufen zu können, besitzt die Funktion `transalt()` einen weiteren, optionalen Parameter `i`, der anfangs den Wert 1 besitzt und bei jedem rekursiven Aufruf um 1 erhöht wird. Hat man die Menge `V` in einer Kette rekursiver Aufrufe vollständig durchlaufen (was genau dann der Fall ist, wenn die Bedingung `(i <= *V)` *nicht* mehr erfüllt ist), so entspricht der Parameter `T` nicht mehr einer ursprünglichen Alternative `T` $\in$ `s/alts`, sondern einer modifizierten Alternative $S \in \vartheta(T)$. Nach Definition des Zustandsübergangs wird in einer solchen Alternative sukzessive jeder erweiterte Zustand $t \in S$ durch seinen transformierten Zustand $t' = \tau_a(t)$ ersetzt, um jeweils eine neue Alternative $S - t + t'$ zu erzeugen. Da die so entstandenen Alternativen in der Menge `A` gespeichert werden sollen, wird hier kein „update in place" vorgenommen, sondern stattdessen eine Kopie `T_ = +T` von `T` erzeugt und modifiziert.

Um die Optimierungsfunktion ρ_1 zu berücksichtigen, werden Alternativen mit einem ungültigen Zustand/Wert-Paar `t_` übergangen.

Anmerkung

Ebenso wie die Definition des *Zustandsmodells* für parallele Quantorausdrücke in § 4.5.7.4 sowie die zugehörige Verifikation in § B.2.4.4 einen „Höhepunkt formaler Komplexität" dieser Arbeit darstellt, stellt zweifellos auch die soeben vorgestellte *Implementierung* der Definitionen einen „Höhepunkt implementierungstechnischer Komplexität" dar,[30] obwohl die Gesamtlänge des Codes nur etwa 75 Zeilen beträgt.

[30] Die Bezeichnung „Komplexität" ist hier nicht formal im Sinne von „Berechnungsaufwand", sondern umgangssprachlich als „Kompliziertheit" zu verstehen.

Aus diesem Grund wäre es nicht verwunderlich, wenn auch nach mehrmaliger Lektüre dieses Abschnitts gewisse Detailfragen offenbleiben. Allerdings kann es nicht Sinn und Zweck dieser Ausführungen sein, ein tiefgehendes und umfassendes Verständnis der gesamten Implementierung zu vermitteln. Vielmehr sollte der vorliegende Abschnitt 4.6 lediglich einen *Einblick* in das zentrale Modul des Programms, die implementierungstechnische Umsetzung der operationalen Semantik von Interaktionsausdrücken, vermitteln.

4.6.5 Hauptprogramm

Abbildung 4.22 zeigt ein einfaches Hauptprogramm, das mit Hilfe der Funktionen `init()`, `trans()` und ggf. `final()` sowohl das für praktische Anwendungen wesentliche *Aktionsproblem* als auch das in § 4.1.1 erwähnte *Wort*- und *Teilwortproblem* löst. Mit Hilfe der Parserfunktion `parse:expr()` wird der als Kommandoargument `args[1]` übergebene (d. h. als Zeichenkette vorliegende) Ausdruck zunächst in einen äquivalenten Operatorbaum `x` vom Typ `Expr` umgewandelt (vgl. § 4.3.4), dessen initialer Zustand `s` mittels der Funktion `init()` bestimmt wird.

```
begin {
    Expr x = parse:expr(args[1]);
    State s = init(x);

    if (*args == 1) {           // Aktionsproblem.
        loop {
            print "Action: ";

            Str str = "";
            input str;
            if (str == "") exit; // Programmende bei leerer Eingabe.
            Action a = parse:action(str);

            if (State s_ = trans(s, a)) s = s_;
            else print "Illegal action (ignored)", newline;
        }
    }
    else if (*args == 2) {      // Wort- und Teilwortproblem.
        Seq(Action) w = parse:word(args[2]);

        Action a;
        forall (a << w) s = trans(s, a);

        if (final(s)) print "Complete word";
        else if (s) print "Partial word";
        else print "Illegal word";
        print newline;
    }
}
```

Abbildung 4.22: Hauptprogramm

Wenn kein weiteres Kommandoargument übergeben wurde, d. h. wenn die Sequenz `args` die Länge 1 besitzt, wird in einer Endlosschleife immer wieder eine Zeichenkette `str` eingelesen und mit Hilfe der Parserfunktion `parse:action()` als Aktion `a` interpretiert. Wenn der Zustandsübergang `trans(s, a)` erfolgreich ist, d. h. einen gültigen Folgezustand `s_` liefert, ersetzt dieser den aktuellen

Zustand `s`; andernfalls wird darauf hingewiesen, daß die zuletzt eingegebene Aktion `a` unzulässig war und daher ignoriert wird.

Wird das Programm mit einem zweiten Kommandoargument `args[2]` aufgerufen, so wird dieses mit Hilfe der Parserfunktion `parse:word()` als Wort `w`, d. h. als Folge von Aktionen `a` interpretiert. Führt man für jede dieser Aktionen einen Zustandsübergang `trans(s, a)` durch, so erhält man letztlich den Folgezustand $\hat{\sigma}_{\mathtt{w}}(\mathtt{x})$ des gegebenen Ausdrucks `x`. Wenn dieser Zustand `s` ein Endzustand ist, stellt das vorliegende Wort `w` gemaß § 4.5.2.4 ein vollständiges Wort von `x` dar; handelt es sich lediglich um einen gültigen Zustand, so ist `w` immerhin ein partielles Wort von `x`; andernfalls stellt `w` eine unzulässige Aktionsfolge des Ausdrucks `x` dar.

4.7 Komplexitätsbetrachtungen

4.7.1 Vorüberlegungen und Definitionen

Betrachtet man die Definition der optimierten Zustandsübergangsfunktion $\hat{\tau}_a(s)$ in § 4.5 bzw. ihre implementierungstechnische Umsetzung `trans()` in § 4.6, so erkennt man, daß ihr Berechnungsaufwand im wesentlichen von der *Größe* des Zustands s abhängt, d. h. von der *Anzahl der Knoten* in einer *baumartigen Repräsentation* dieses Zustands (vgl. Abb. 4.23). Die Zustandsübergangsfunktion wiederum stellt das Kernstück der Algorithmen zur Lösung des Wort-, Teilwort- und Aktionsproblems dar (vgl. § 4.6.5). Um daher Aussagen über die *Komplexität* dieser Algorithmen treffen zu können, soll im folgenden untersucht werden, wie groß die Zustände $\hat{\sigma}_w(x)$ eines Ausdrucks x unter bestimmten Voraussetzungen werden können.

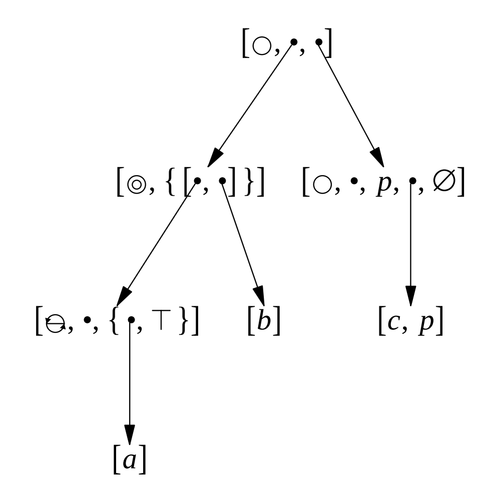


Abbildung 4.23: Baumartige Repräsentation des Zustands $\sigma(x)$ für $x \equiv ((\ominus\, a) \circledcirc b) \circ \bigcirc_p c(p)$

4.7.1.1 Tiefe von Zustandsbäumen

Aus den Definitionen der Abschnitte 4.5.4 und 4.5.7 folgt, daß die *Tiefe* (oder Höhe) $\delta(s)$ eines Zustandsbaums $s = \hat{\sigma}_w(x)$, d. h. die maximale Länge (Kantenzahl) eines Pfads von der Wurzel bis zu einem Blattknoten, durch eine Konstante $\mathrm{D}(x)$ begrenzt ist, die lediglich vom Ausdruck x, nicht jedoch vom Wort w abhängt. Konkret gilt für $w \in \Sigma^*$:

$$\delta\big(\hat{\sigma}_w(x)\big) \le \mathrm{D}(x) = \begin{cases} 0 & \text{für } x \equiv a, \\ \mathrm{D}(y) + 1 & \text{für } x \equiv \bigcirc\, y \text{ oder } x \equiv \underset{p}{\bigcirc}\, y, \\ \max(\mathrm{D}(y), \mathrm{D}(z)) + 1 & \text{für } x \equiv y \bigcirc z, \end{cases}$$

d. h. $\mathrm{D}(x)$ entspricht der *maximalen Verschachtelungstiefe* von Operatoren im Ausdruck x.

4.7.1.2 Verzweigungsgrad und Größe von Zustandsbäumen. Klassifikation von Ausdrücken

Der *Verzweigungsgrad* (oder einfach Grad) eines einzelnen *Knotens* entspricht der Anzahl der direkten *Nachfolger* dieses Knotens, während der Verzweigungsgrad eines ganzen *Baums* als *maximaler Verzweigungsgrad* seiner Knoten definiert ist. Offensichtlich besitzt ein Baum mit der Tiefe d und dem Verzweigungsgrad b höchstens

$$1 + b + b^2 + \ldots + b^d = \mathrm{O}\big(b^d\big)$$

Knoten, d. h. seine *Größe* ist von der Größenordnung b^d. Da die Tiefe $\delta(s)$ eines Zustandsbaums $s = \hat{\sigma}_w(x)$ durch die Konstante $\mathrm{D}(x)$ begrenzt ist, hängt seine Größe $\gamma(s)$ somit im wesentlichen von seinem Verzweigungsgrad $\beta(s)$ ab:

1. Sofern $\beta(s)$ *konstant* ist, d. h. wenn $\beta(s) \le \mathrm{B}(x)$ für eine Konstante $\mathrm{B}(x)$ gilt, die nur vom Ausdruck x, nicht jedoch von der Länge $n = |w|$ des Wortes w abhängt, so ist auch $\gamma(s)$ durch eine Konstante $\mathrm{G}(x)$ beschränkt, die nur vom Ausdruck x abhängt:

   $$\gamma(s) = \mathrm{O}(\mathrm{G}(x)) \quad \text{mit} \quad \mathrm{G}(x) = (\mathrm{B}(x))^{\mathrm{D}(x)}.$$

   In diesem Fall heißt der Ausdruck x *harmlos*, weil der Berechnungsaufwand für einen Zustandsübergang konstant ist.

2. Wenn $\beta(s)$ *polynomiell beschränkt* bzgl. n ist, d. h. wenn $\beta(s) = \mathrm{O}\big(n^k\big)$ für ein $k \in \mathbb{N}$ gilt, so gilt dies auch für $\gamma(s)$:

   $$\gamma(s) = \mathrm{O}\Big(\big(n^k\big)^{\mathrm{D}(x)}\Big) = \mathrm{O}\big(n^{k\mathrm{D}(x)}\big) = \mathrm{O}\big(n^{\tilde{k}}\big) \quad \text{für} \quad \tilde{k} = k \cdot \mathrm{D}(x) \in \mathbb{N}.$$

   In diesem Fall heißt der Ausdruck x *gutartig*, weil der Berechnungsaufwand für einen Zustandsübergang polynomiell beschränkt ist.

3. Wenn $\beta(s)$ *nicht polynomiell beschränkt* ist, d. h. wenn $\beta(s) \ne \mathrm{O}\big(n^k\big)$ für alle $k \in \mathbb{N}$ gilt, so gilt dies auch für $\gamma(s)$:

   $$\gamma(s) \ne \mathrm{O}\big(n^k\big) \quad \text{für alle } k \in \mathbb{N}.$$

   In diesem Fall heißt der Ausdruck x *bösartig*, weil der Berechnungsaufwand für einen Zustandsübergang nicht polynomiell beschränkt ist.

4.7.1.3 Klassifikation von Operatoren

Wie in den nachfolgenden Abschnitten erläutert wird, hängt der Verzweigungsgrad $\beta(s)$ eines Zustandsbaums $s = \hat{\sigma}_w(x)$ (bzw. der Verzweigungsgrad seiner einzelnen Knoten) in erster Linie von den im Ausdruck x auftretenden *Operatoren* und eventuell (je nach Operator) auch von den zugehörigen *Operanden* (d. h. Teilausdrücken des Operators) ab. Abhängig davon, welchen Verzweigungsgrad der *Wurzelknoten* eines Zustandsbaums $\hat{\sigma}_w(x)$ für einen Ausdruck $x \equiv \bigcirc\, y$ bzw. $x \equiv y \bigcirc z$ bzw. $x \equiv \underset{p}{\bigcirc}\, y$ besitzt, wird der Operator $\bigcirc$ daher ebenfalls einer der drei „Güteklassen“ harmlos, gutartig oder bösartig zugeordnet:

1. Wenn der Verzweigungsgrad des Wurzelknotens *konstant* ist, heißt der Operator *harmlos*.
2. Wenn der Verzweigungsgrad polynomiell beschränkt bzgl. der Länge $n = |w|$ des Wortes w ist, heißt der Operator *gutartig*.
3. Andernfalls, d. h. wenn der Verzweigungsgrad nicht polynomiell beschränkt ist, heißt der Operator *bösartig*.

4.7.1.4 Zusammenhang zwischen Operatoren und Ausdrücken

Offensichtlich hängt die Klassifikation eines Ausdrucks wie folgt mit der Klassifikation seiner Operatoren und Teilausdrücke zusammen:

1. Ein elementarer Ausdruck $x \equiv \bigcirc\, y$ oder $x \equiv y \bigcirc z$ ist *harmlos* (bzw. *gutartig*), wenn sowohl der Operator $\bigcirc$ als auch die Teilausdrücke y und ggf. z harmlos (bzw. gutartig) sind.
   Andernfalls, d. h. wenn der Operator $\bigcirc$ oder einer der Teilausdrücke y oder ggf. z bösartig ist, ist der Ausdruck *bösartig*.
2. Ein Quantorausdruck $x \equiv \underset{p}{\bigcirc}\, y$ ist *harmlos* (bzw. *gutartig*), wenn sowohl der Operator $\bigcirc$ als auch der abstrakte Zweig y und sämtliche konkretisierten Zweige y_p^ω harmlos (bzw. gutartig) sind.
   Andernfalls, d. h. wenn der Operator $\bigcirc$ oder der abstrakte Zweig y oder einer der konkretisierten Zweige y_p^ω bösartig ist, ist der Ausdruck *bösartig*.

4.7.1.5 Zustandszahl

Die *Zustandszahl* $\eta(x)$ eines Ausdrucks x ist definiert als die Anzahl der *verschiedenen* Zustände $\hat{\sigma}_w(x)$ des Ausdrucks x, die man erhält, wenn w alle Worte der Menge Σ^* durchläuft:

$$\eta(x) = \left|\hat{\Theta}(x)\right| = \left|\{\, \hat{\sigma}_w(x) \mid w \in \Sigma^* \,\}\right|.$$

4.7.2 Teilklassen von Ausdrücken

Da beliebige Interaktionsausdrücke, wie in § 4.7.3.3 gezeigt werden wird, *potentiell bösartig* sind, besteht das Ziel der folgenden Abschnitte darin, möglichst große und praktisch relevante *Teilklassen* von Ausdrücken zu identifizieren, für die gezeigt werden kann, daß sie gutartig oder sogar harmlos sind.

4.7.2.1 Quasi-reguläre Ausdrücke

Definition

Gemäß § 3.5.2 stellen die Operatoren für Konjunktion, Synchronisation und parallele Komposition keine Erweiterung der Ausdrucksmächtigkeit regulärer Ausdrücke dar. Daher werden Interaktionsausdrücke, die weder Quantoren noch parallele Iterationen enthalten (wobei die parallele Iteration als spezieller Quantor aufgefaßt werden kann, vgl. § 3.4.11), im folgenden als *quasi-reguläre Ausdrücke* bezeichnet.

Anmerkung: Quasi-reguläre Ausdrücke können nichtsdestotrotz *parametrisiert* sein, d. h. zum Beispiel konstante Parameterwerte enthalten oder im *Wirkungsbereich* von Quantoren stehen. Die Definition verlangt lediglich, daß die Ausdrücke *selbst* weder Quantorausdrücke noch parallele Iterationen enthalten.

Satz

Ein quasi-regulärer Ausdruck x besitzt nur endlich viele verschiedene Zustände, d. h. es gilt $\eta(x) < \infty$.

Beweis

1. Für einen *atomaren Ausdruck* $x \equiv b$ gibt es genau drei verschiedene Zustände: b, $\top$ und $\bot$.
2. Ein Zustand eines *Booleschen Ausdrucks* $x \equiv y \circledcirc z$ mit $\circledcirc \in \{ \bigcirc, \mathbf{o}, \bullet \}$ und quasi-regulären Teilausdrücken y und z ist im wesentlichen ein Paar von Teilzuständen l und r, von denen es nach Induktionsvoraussetzung $\eta(y)$ bzw. $\eta(z)$ verschiedene gibt. Daher besitzt der Ausdruck x höchstens $\eta(y)\,\eta(z)$ verschiedene Zustände.
3. Ein Zustand einer *sequentiellen Komposition* $x \equiv y - z$ besteht im wesentlichen aus einem Teilzustand des linken Teilausdrucks y, von denen es nach Induktionsvoraussetzung $\eta(y)$ verschiedene gibt, sowie einer Menge R von Teilzuständen des rechten Teilausdrucks z, von denen es $\eta(z)$ verschiedene gibt. Daher gibt es höchstens $2^{\eta(z)}$ verschiedene Zustandsmengen R und somit höchstens $\eta(y) \cdot 2^{\eta(z)}$ verschiedene Zustände des Ausdrucks x.
   Analog ergibt sich, daß eine *sequentielle Iteration* $x \equiv \ominus\, y$ höchstens $2^{\eta(y)}$ verschiedene Zustände besitzt.
4. Ein Zustand einer parallelen Komposition $x \equiv y \circledcirc z$ besteht aus einer Menge von Paaren von Teilzuständen l und r, von denen es höchstens $2^{\eta(y)\eta(z)}$ verschiedene gibt.

Anmerkungen

1. Da die Quasi-Regularität eines Ausdrucks x eine rein syntaktische Eigenschaft darstellt, ist sie unabhängig von irgendwelchen Parameterbelegungen. Daher ist ein konkretisierter Ausdruck x_p^ω genau dann quasi-regulär, wenn der abstrakte Ausdruck x diese Eigenschaft besitzt.
2. Für beliebige Interaktionsausdrücke x gilt die Aussage $\eta(x) < \infty$ nicht. Als Gegenbeispiel betrachte man den Ausdruck $x \equiv \circledcirc\,(a - b)$ sowie Worte $w_n = \langle a, \ldots, a \rangle$ der Länge $n \in \mathbb{N}$. Da der Ausdruck nach Verarbeitung von w_n genau n b's akzeptiert, müssen zwei Zustände $\hat{\sigma}_{w_m}(x)$ und $\hat{\sigma}_{w_n}(x)$ für $m \neq n$ verschieden sein, d. h. es gibt unendlich viele verschiedene Zustände $\hat{\sigma}_w(x)$ des Ausdrucks x.

4.7.2.2 Vollständig und homogen quantifizierte Ausdrücke

Definitionen

1. Ein Quantorausdruck $x \equiv \underset{p}{\bigcirc}\, y$ (bzw. der Quantorrumpf y) heißt *vollständig quantifiziert* (bzgl. des Parameters p), wenn jede Aktion $b \in \alpha(y)$ des Quantorrumpfs y vom Parameter p *abhängt*, d. h. p in ihrer Parameterliste enthält.
2. Der Ausdruck heißt *homogen quantifiziert*, wenn der Parameter p in *gleichartigen* Aktionen (d. h. Aktionen mit derselben *Signatur*) immer an *denselben Positionen* auftritt, d. h. wenn für zwei Aktionen $b = [b_0, b_1, \ldots, b_n]$ und $b' = [b_0, b'_1, \ldots, b'_n]$ aus dem Alphabet von y stets gilt:

$$\{ i \in \mathbb{N} \mid b_i = p \} = \{ i \in \mathbb{N} \mid b'_i = p \}.$$

Beispiele

1. Der Ausdruck $x_1 \equiv \underset{p}{\circledcirc}\,(a(p) - b)$ ist homogen, aber nicht vollständig quantifiziert, weil die Aktion b *nicht* vom Parameter p abhängt.

2. Der Ausdruck $x_2 \equiv \underset{p}{\circledcirc} \left(a(\omega_1,\, p) - a(p,\, \omega_2)\right)$ ist vollständig, aber nicht homogen quantifiziert, weil der Parameter p in den gleichartigen Aktionen $a(\omega_1,\, p)$ und $a(p,\, \omega_2)$ an *verschiedenen* Positionen auftritt.
3. Der Ausdruck $x_3 \equiv \underset{p}{\circledcirc} \left(a(p,\, \omega_1) - a(p,\, \omega_2)\right)$ ist sowohl vollständig als auch homogen quantifiziert.

Satz

Gegeben sei ein Quantorausdruck $x \equiv \underset{p}{\bigcirc}\, y$.

1. Wenn der Ausdruck x *homogen* quantifiziert ist, besitzt jede Aktion $a \in \Sigma$ *höchstens einen* relevanten Parameterwert, d. h. für alle $a \in \Sigma$ gilt $\left|\Omega_a(y,\, p)\right| \leq 1$.
2. Wenn der Ausdruck x *vollständig* quantifiziert ist, akzeptiert der abstrakte Zweig y *keine* konkreten Aktionen, d. h. für jedes Wort $w \neq \langle\rangle$ ist der Zustand $\hat{\sigma}_w(y)$ *ungültig*.
3. Wenn der Ausdruck x *vollständig und homogen* quantifiziert ist, sind die Alphabete der konkretisierten Zweige y_p^ω *paarweise disjunkt*, d. h. es gilt $\alpha\left(y_p^\omega\right) \cap \alpha\left(y_p^\pi\right) = \varnothing$ für $\omega \neq \pi$.

Beweis

1. Gemäß § 4.5.6.2 gehört ein Wert $\omega \in \Omega$ genau dann zur Menge $\Omega_a(y,\, p)$ der relevanten Parameterwerte von a, wenn der Quantorrumpf y eine Aktion $\tilde{a} \neq a$ enthält, die durch die Konkretisierung $\tilde{a}_p^\omega$ gleich a wird. Insbesondere müssen daher alle derartigen Aktionen $\tilde{a}$ dieselbe Signatur wie a besitzen und somit auch untereinander gleichartig sein.
   Aufgrund der homogenen Quantifizierung des Ausdrucks x tritt der Parameter p in diesen Aktionen somit immer an denselben Positionen $i,\, j,\, \ldots$ auf, d. h. es kann höchstens *einen* Wert $\omega = a_i = a_j = \ldots$ geben, für den $\tilde{a}_p^\omega = a$ gilt.
2. Wenn der Ausdruck x vollständig quantifiziert ist, so gilt offensichtlich $\alpha'(y) = \alpha(y) \cap \Sigma = \varnothing$, da jede Aktion $b \in \alpha(y)$ den Parameter p als Argument enthält und somit nicht zur Menge Σ der konkreten Aktionen gehört. Aufgrund der Inklusion $\Psi(y) \subseteq \alpha'(y)^* = \{\,\langle\rangle\,\}$ (vgl. § 3.4.4) kann der Ausdruck y daher keine nicht-leeren partiellen Worte $w \in \Sigma^*$ besitzen, und somit ist der Zustand $\hat{\sigma}_w(y)$ für $w \neq \langle\rangle$ ungültig.
3. Wenn zwei Aktionen $a_p^\omega \in \alpha\left(y_p^\omega\right)$ und $b_p^\pi \in \alpha\left(y_p^\pi\right)$ gleich sein sollen, müssen die Aktionen $a,\, b \in \alpha(y)$ dieselbe Signatur besitzen. Aufgrund der vollständigen und homogenen Quantifizierung des Ausdrucks x enthalten a und b beide den Parameter p, und zwar an denselben Positionen $i,\, j,\, \ldots$ Damit dann $a_p^\omega = b_p^\pi$ gelten kann, muß offensichtlich $\omega = \pi$ sein.
   Daraus folgt durch Umkehrschluß, daß die Alphabete $\alpha\left(y_p^\omega\right)$ und $\alpha\left(y_p^\pi\right)$ für $\omega \neq \pi$ disjunkt sind.

Anmerkungen

1. Auch die vollständige oder homogene Quantifizierung eines Ausdrucks ist eine rein syntaktische Eigenschaft, die nicht von Parameterbelegungen abhängt. Daher ist ein konkretisierter Ausdruck x_q^π genau dann vollständig und/oder homogen quantifiziert, wenn der abstrakte Ausdruck x die entsprechende(n) Eigenschaft(en) besitzt.
2. Die dritte Behauptung des Satzes ist falsch, wenn eine der beiden Voraussetzungen fehlt:
   Wenn ein Ausdruck nicht vollständig quantifiziert ist (wie z. B. oben der Beispielausdruck x_1), so gehören Aktionen, die nicht vom Parameter p abhängen (im Beispiel die Aktion b), zu *allen* Zweigen des Quantorausdrucks.
   Wenn ein Ausdruck nicht homogen quantifiziert ist (wie z. B. der Beispielausdruck x_2), so können verschiedene Zweige des Quantorausdrucks gemeinsame Aktionen besitzen. Im Beispiel gehört die Aktion $a(\omega_1,\, \omega_2)$ sowohl zum Zweig $y_p^{\omega_1}$ als auch zum Zweig $y_p^{\omega_2}$.

4.7.2.3 Injektive und fokussierte Ausdrücke

Definitionen

1. Ein Ausdruck x heißt *injektiv*, wenn es zu jeder Aktion $a \in \Sigma$ höchstens *einen* Zustand $s \in \hat{\Theta}(x)$ gibt, der diese Aktion erfolgreich verarbeiten kann, d. h. für den der Folgezustand $s' = \hat{\tau}_a(s)$ *gültig* ist.
2. Der Ausdruck x heißt *initial fokussiert*, wenn für alle $w \neq \langle\rangle$ gilt:

   $\hat{\sigma}_w(x) \neq \sigma(x)$,

   d. h. wenn alle echten Folgezustände verschieden vom initialen Zustand des Ausdrucks sind.
3. Der Ausdruck x heißt *terminal fokussiert*, wenn er höchstens den Zustand $\top$ als Endzustand besitzt.

Ein Ausdruck, der sowohl initial als auch terminal fokussiert ist, wird auch als *beidseitig fokussiert* bezeichnet.

Anmerkung: Die Definition eines injektiven Ausdrucks schließt nicht aus, daß es für ein und denselben Zustand s verschiedene Aktionsfolgen $w_1, w_2, \ldots \in \Sigma^*$ gibt, die alle in diesen Zustand führen, d. h. für die $s = \hat{\sigma}_{w_1}(x) = \hat{\sigma}_{w_2}(x) = \ldots$ gilt.

Anschauliche Interpretation

Die Begriffe der Injektivität und Fokussierung eines Ausdrucks x lassen sich am besten anhand eines *Zustandsübergangs-Diagramms* veranschaulichen, in dem jeder Endzustand von x durch einen *schwarzen Kreis*, jeder sonstige Zustand $s \in \hat{\Theta}(x)$ durch einen *weißen Kreis* und jeder *gültige* Zustandsübergang $s' = \hat{\tau}_a(s)$ durch eine gerichtete und mit a beschriftete *Kante* von s nach s' dargestellt wird:

1. Ein injektiver Ausdruck ist dadurch charakterisiert, daß die Kanten seines Zustandsübergangs-Diagramms *paarweise verschiedene Beschriftungen* tragen (vgl. Abb. 4.24).
2. Ein initial fokussierter Ausdruck zeichnet sich dadurch aus, daß sein initialer Zustand $\sigma(x)$ im Zustandsübergangs-Diagramm eine *Quelle* darstellt, d. h. einen Knoten, der *keine eingehenden Kanten* besitzt (vgl. Abb. 4.25).
3. Ein terminal fokussierter Ausdruck schließlich ist dadurch gekennzeichnet, daß der spezielle Zustand $\top$ (der aufgrund seiner Definition stets eine *Senke* darstellt, d. h. einen Knoten, der *keine ausgehenden Kanten* besitzt) den einzigen schwarzen Kreis im Zustandsübergangs-Diagramm darstellt (vgl. ebenfalls Abb. 4.25).

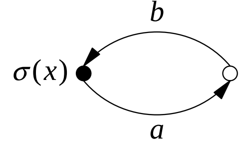


Abbildung 4.24: Zustandsübergangs-Diagramm des injektiven Ausdrucks $x \equiv \ominus (a - b)$

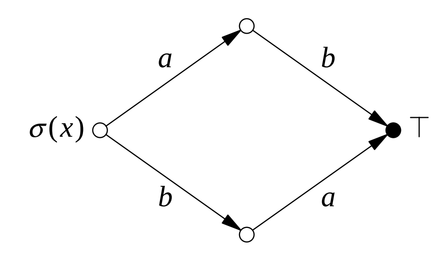


Abbildung 4.25: Zustandsübergangs-Diagramm
des beidseitig fokussierten Ausdrucks $x \equiv (a - b) \bigcirc (b - a)$

Satz

1. Ein *atomarer Ausdruck* $x \equiv b$ ist injektiv und beidseitig fokussiert.
2. Eine *sequentielle Komposition* $x \equiv y - z$ ist injektiv, wenn die Teilausdrücke y und z injektiv und disjunkt sind und der Teilausdruck y terminal fokussiert ist.
   Ist der Teilausdruck y darüber hinaus initial und/oder der Teilausdruck z terminal fokussiert, so besitzt auch der Gesamtausdruck x diese Eigenschaften.
   Insbesondere ist eine Sequenz $a_1 - \ldots - a_n$ von *paarweise verschiedenen* Aktionen $a_1, \ldots, a_n$ injektiv und beidseitig fokussiert.
3. Eine *sequentielle Iteration* $x \equiv \ominus\, y$ ist injektiv, wenn ihr Rumpf y injektiv und beidseitig fokussiert ist.
4. Eine *Disjunktion* $x \equiv y \bigcirc z$ ist injektiv und initial fokussiert, wenn die Teilausdrücke y und z diese Eigenschaften besitzen und ihre Alphabete disjunkt sind.
   Sind beide Teilausdrücke darüber hinaus terminal fokussiert, so besitzt auch der Gesamtausdruck diese Eigenschaft.
5. Eine *Option* $x \equiv \smile y$ ist injektiv und initial fokussiert, wenn ihr Rumpf y diese Eigenschaften besitzt.
6. Ein *Disjunktions-Quantorausdruck* $x \equiv \underset{p}{\bigcirc}\, y$ ist injektiv und initial fokussiert, wenn alle konkretisierten Zweige y_p^ω diese Eigenschaften besitzen und der Ausdruck vollständig und homogen quantifiziert ist.
   Sind alle konkretisierten Zweige y_p^ω darüber hinaus terminal fokussiert, so besitzt auch der Gesamtausdruck diese Eigenschaft.

Da der Beweis dieses Satzes vergleichsweise umfangreich ist, wurde er in Anhang B (§ B.4) ausgelagert.

Anmerkung

Anders als die Quasi-Regularität (§ 4.7.2.1) und die vollständige oder homogene Quantifizierung (§ 4.7.2.2), hängt die Injektivität und Fokussierung eines Ausdrucks u. U. von konkreten Parameterbelegungen ab. Beispielsweise ist der Ausdruck $x \equiv a(p) - a(q)$ mit $p \neq q \in \Pi$ gemäß Behauptung 2 injektiv und beidseitig fokussiert, da die Aktionen $a(p)$ und $a(q)$ verschieden sind. Dasselbe gilt auch für den konkretisierten Ausdruck $x_{p,\,q}^{\omega,\,\pi} \equiv a(\omega) - a(\pi)$, wenn die Werte ω und π verschieden sind. Sind die Werte jedoch gleich, so erfüllt der resultierende Ausdruck $x_{p,\,q}^{\omega,\,\omega} \equiv a(\omega) - a(\omega)$ die Voraussetzung von Behauptung 2 nicht mehr. Tatsächlich ist dieser Ausdruck nicht mehr injektiv, da er sowohl im initialen Zustand als auch nach erfolgreicher Verarbeitung der ersten Aktion die Aktion $a(\omega)$ akzeptiert. Ebenso lassen sich Ausdrücke formulieren, deren initiale oder terminale Fokussierung abhängig von bestimmten Parameterbelegungen ist.

Aus diesen Gründen muß bei Argumentationen, die sich auf die Injektivität oder Fokussierung eines (Teil-)Ausdrucks stützen, sorgfältig darauf geachtet werden, daß diese Eigenschaften für alle in Frage kommenden Parameterbelegungen, d. h. für alle entsprechenden Konkretisierungen des Ausdrucks, erfüllt sind. Dies betrifft insbesondere die Aussagen zur parallelen Iteration in § 4.7.3.4 sowie zu parallelen Multiplikatoren in Tab. 4.27 (§ 4.7.5.2), die in den Beispielen 1 und 2 von § 4.7.5.3 verwendet werden.

4.7.3 Aussagen zu elementaren Operatoren

Nach den vorangegangenen Vorbereitungen ist es nun möglich, konkrete Komplexitätsaussagen sowohl zu elementaren Operatoren als auch zu Quantoren (§ 4.7.4) zu formulieren.

4.7.3.1 Boolesche Operatoren

Satz

Die Booleschen Operatoren Disjunktion (einschließlich Option), Konjunktion und Synchronisation sind *harmlos*.

Beweis

Der initiale Zustand eines Booleschen Ausdrucks $x \equiv y \bigcirc z$ besitzt den Verzweigungsgrad 2, der bei nachfolgenden Zustandsübergängen entweder unverändert bleibt oder durch Optimierungen reduziert wird, wenn ein disjunktiver Zustand durch einen seiner Teilzustände oder ein kompletter Zustand durch den Zustand $\perp$ ersetzt wird (vgl. § 4.5.4.2 bis § 4.5.4.5).

Folgerungen

Diese Tatsache ist insbesondere für die häufig verwendete *modulare Kombination* von Ausdrücken mit Hilfe des *Synchronisationsoperators* bedeutsam, weil sie besagt, daß die Komplexität eines zusammengesetzten Ausdrucks $x_1 \boldsymbol{\circ} \ldots \boldsymbol{\circ} x_n$ einzig und allein von der Komplexität der Teilausdrücke x_i abhängt und durch die Verknüpfung mittels Synchronisation keine zusätzliche Komplexität entsteht.

4.7.3.2 Sequentielle Operatoren

Satz

1. Die Operatoren für sequentielle Komposition und Iteration sind *gutartig*.
2. Sofern der linke Teilausdruck y einer sequentiellen Komposition $y - z$ bzw. der Rumpf y einer sequentiellen Iteration $\ominus\, y$ terminal fokussiert ist, sind die Operatoren sogar *harmlos*.
3. Die Operatoren sind auch *harmlos*, wenn der rechte Teilausdruck z einer sequentiellen Komposition $y - z$ bzw. der Rumpf y einer sequentiellen Iteration $\ominus\, y$ quasi-regulär ist.
4. Wenn die Alphabete der Teilausdrücke y und z disjunkt sind, ist der sequentielle Kompositionsoperator ebenfalls *harmlos*.

Beweis

1. Der Verzweigungsgrad eines Zustands $s = [-, z, l, R]$ (sequentielle Komposition) bzw. $s = [\ominus, y, T]$ (sequentielle Iteration) wird im wesentlichen durch die Kardinalität der Zustandsmenge R bzw. T bestimmt. Für einen initialen Zustand $s = \sigma(x)$ enthält diese Menge maximal ein

bzw. zwei Elemente, und bei jedem Zustandsübergang $\hat{\tau}_a(s)$ kommt maximal ein Element hinzu (vgl. § 4.5.4.6 und § 4.5.4.7). Daher wächst die Kardinalität dieser Menge − und somit auch der Verzweigungsgrad der Zustände $\hat{\sigma}_w(x)$ − höchstens linear bzgl. der Länge $n = |w|$ des Wortes w.

2. Ist der linke Teilausdruck y einer sequentiellen Komposition $y - z$ terminal fokussiert, so bleibt die Zustandsmenge R solange leer, bis der Teilausdruck y seinen Endzustand $\top$ erreicht. Der dann erreichte Gesamtzustand $s = [-, z, \top, \{ \sigma(z) \}]$ wird durch die Optimierung ρ_2 durch den Teilzustand $\sigma(z)$ ersetzt und besitzt ab diesem Zeitpunkt quasi den konstanten Verzweigungsgrad 1.
   Ist der Rumpf y einer sequentiellen Iteration $\ominus\, y$ terminal fokussiert, so bleibt die Kardinalität der Zustandsmenge T bei einem Zustandsübergang entweder konstant, oder sie erhöht sich um 1, wenn der Endzustand $\top$ des Rumpfs y erreicht wird. Beim nächsten Zustandsübergang geht dieser Teilzustand $\top$ in den ungültigen Zustand $\bot$ über, der durch die Optimierung ρ_1 entfernt wird. Da sich die Kardinalität von T somit wieder um 1 verringert, kann sie insgesamt nicht größer als 2 werden.

3. Da die Menge R (bzw. T) nur Zustände des rechten Teilausdrucks z (bzw. des Iterationsrumpfs y) enthält, ist ihre Kardinalität außerdem durch die Anzahl $\eta(z)$ (bzw. $\eta(y)$) beschränkt, die für einen quasi-regulären Ausdruck z (bzw. y) eine endliche Konstante darstellt, die nicht von der Länge n abhängt.

4. Wenn die Alphabete der Teilausdrücke y und z disjunkt sind, so folgt durch ähnliche Überlegungen wie in § B.4.2, daß die Kardinalität der Menge R nicht größer als 1 werden kann.

Folgerungen

Die erste Behauptung des Satzes besagt, daß die in praktischen Anwendungen sehr häufig auftretenden sequentiellen Operatoren grundsätzlich bedenkenlos angewandt werden können.

Zusammen mit der Aussage über Boolesche Operatoren (§ 4.7.3.1) impliziert die zweite Behauptung, daß *reguläre Ausdrücke* harmlos sind und somit *sehr effizient* (d. h. mit *konstantem* Berechnungsaufwand für einen Zustandsübergang) verarbeitet werden. Daher läßt sich das Wortproblem für reguläre Ausdrücke in *linearer Zeit* lösen, d. h. die Implementierung regulärer Ausdrücke mit Hilfe hierarchischer Zustände ist zwar „komplizierter“, aber nicht komplexer als eine Implementierung durch endliche Automaten [Hopcroft90, Schöning95].

Anmerkung

Die sequentiellen Operatoren sind nicht grundsätzlich harmlos. Als Gegenbeispiele betrachte man die Ausdrücke

$$x_1 \equiv y_1 - z_1 \quad \text{mit} \quad y_1 \equiv \ominus\, a \quad \text{und} \quad z_1 \equiv \circledcirc\, (a - b)$$

und

$$x_2 \equiv \ominus\, z_2 \quad \text{mit} \quad z_2 \equiv \circledcirc\, (a - \smile b)$$

sowie Worte $w_n = \langle a, \ldots, a \rangle$ der Länge n. In beiden Beispielen wächst die Kardinalität der Zustandsmenge R bzw. T tatsächlich linear bezüglich n.[31]

4.7.3.3 Parallele Komposition

Satz

1. Der parallele Kompositionsoperator ist *potentiell bösartig*, d. h. es gibt Ausdrücke $x \equiv y \circledcirc z$, für die der Verzweigungsgrad ihrer Zustände $\hat{\sigma}_w(x)$ *exponentiell* mit der Länge $n = |w|$ des Wortes w wächst.

[31] Aus Platzgründen werden derartige Gegenbeispiele im folgenden lediglich erwähnt und nicht näher erläutert.

2. Wenn die Teilausdrücke y und z quasi-regulär sind, oder wenn ihre Alphabete disjunkt sind, ist der parallele Kompositionsoperator jedoch harmlos.

Beweis

1. Der initiale Zustand $\sigma(x) = [\circledcirc, A]$ einer parallelen Komposition $x \equiv y \circledcirc z$ mit $A = \{ [\sigma(y), \sigma(z)] \}$ besitzt den Verzweigungsgrad 2. Beim einfachen Zustandsübergang $\tau_a(s)$ eines Zustands $s = [\circledcirc, A]$ wird jede Alternative $[l, r] \in A$ durch *zwei* neue Alternativen $[l', r]$ und $[l, r']$ mit $l' = \tau_a(l)$ und $r' = \tau_a(r)$ ersetzt, d. h. die Kardinalität der Menge A' des transformierten Zustands $s' = [\circledcirc, A']$ ist potentiell doppelt so groß wie die Kardinalität von A. Nach n Zustandsübergängen besitzt die Menge A daher potentiell eine Kardinalität von 2^n und der zugehörige Zustand daher einen Verzweigungsgrad von $2 \cdot 2^n$.
   Aufgrund der Optimierungsfunktion ρ_1 tritt diese prinzipiell mögliche Verdopplung des Verzweigungsgrads bei einem optimierten Zustandsübergang $\hat{\tau}_a(s)$ aber nur dann wirklich ein, wenn *jeder* Teilzustand l bzw. r, der in einer Alternative $[l, r] \in A$ vorkommt, die Aktion a akzeptiert, d. h. einen *gültigen* Folgezustand l' bzw. r' liefert. Dies ist beispielsweise für den Ausdruck $x \equiv y \circledcirc y$ mit $y \equiv \bigcircledcirc_p a(p)$ und ein Wort $w = \langle a(\omega_1), \ldots, a(\omega_n) \rangle$ mit paarweise verschiedenen Werten $\omega_1 \neq \ldots \neq \omega_n \in \Omega$ der Fall.
2. Wenn die Alphabete der Teilausdrücke y und z jedoch disjunkt sind, wird eine Aktion a immer nur von *einem* Teilzustand einer Alternative $[l, r] \in A$ akzeptiert. Dies hat zur Folge, daß die Kardinalität der Menge A konstant den Wert 1 und der Verzweigungsgrads des Ausdrucks somit konstant den Wert 2 behält.
   Wenn die Teilausdrücke y und z beide quasi-regulär sind, gibt es maximal $\eta(y)\,\eta(z)$ verschiedene Alternativen $[l, r]$, d. h. auch in diesem Fall bleibt die Kardinalität der Menge A und somit der Verzweigungsgrad der Zustände durch eine Konstante beschränkt.

Folgerungen

Die zweite Behauptung des Satzes besagt, daß die parallele Komposition − obwohl potentiell bösartig − in vielen praktisch relevanten Fällen harmlos ist. Insbesondere folgt (zusammen mit den Aussagen über Boolesche und sequentielle Operatoren), daß nicht nur reguläre, sondern auch *quasi-reguläre* Ausdrücke harmlos sind, d. h. daß für die Erweiterung regulärer Ausdrücke um Konjunktion, Synchronisation und parallele Komposition kein „Komplexitätspreis“ bezahlt werden muß.

Anmerkungen

Obwohl die erste Behauptung des Satzes prinzipiell „deprimierend“ ist, ist bisher kein *praktisch relevantes* Beispiel eines Ausdrucks bekannt, bei dem sich die parallele Komposition wirklich bösartig verhält. Selbst wenn die Voraussetzungen der zweiten Behauptung nicht erfüllt sind, erweist sich die parallele Komposition in vielen Fällen dennoch als harmlos, d. h. die genannten Voraussetzungen sind zwar hinreichend, aber offensichtlich nicht notwendig.

Darüber hinaus ist interessant, daß sich der im Beweis erwähnte bösartige Ausdruck

$$x \equiv \left(\bigcircledcirc_p a(p) \right) \circledcirc \left(\bigcircledcirc_p a(p) \right)$$

durch eine einfache Äquivalenztransformation (vgl. § 3.4.9) in den gutartigen Ausdruck

$$x' \equiv \bigcircledcirc_p (a(p) \circledcirc a(p))$$

umformen läßt. (Die Gutartigkeit von x' folgt aus § 4.7.4.2.)

Dies wirft die Frage auf, ob es überhaupt *inhärent bösartige* Ausdrücke gibt, d. h. Ausdrücke x, für die *jeder* zu x äquivalente Ausdruck x' bösartig ist. Wenn dies nicht der Fall ist − oder wenn sich bösartige Ausdrücke zumindest häufig in äquivalente gutartige Ausdrücke transformieren lassen −, könnte es lohnend sein, die (manuelle oder automatische) *Optimierung* von Interaktionsausdrücken genauer zu untersuchen, d. h. Strategien und ggf. Algorithmen zu entwickeln, mit deren Hilfe ein gegebener Ausdruck x in einen effizienter implementierbaren Ausdruck x' transformiert werden kann. Im Rahmen dieser Arbeit wurde diese Frage jedoch nicht weiter verfolgt.

4.7.3.4 Parallele Iteration

Satz

1. Der parallele Iterationsoperator ist *gutartig*, wenn sein Rumpf quasi-regulär oder aber injektiv und initial fokussiert ist.
2. Ist der Rumpf *sowohl* quasi-regulär *als auch* injektiv und initial fokussiert, so ist der parallele Iterationsoperator sogar *harmlos*.

Beweis

Eine parallele Iteration $x \equiv ⊚\, y$ wird als paralleler Quantorausdruck $x' \equiv \underset{p}{◎}\, y'$ mit $y' \equiv \lrcorner\, y$ implementiert (vgl. § 3.4.11, § 4.3.1 und § 4.5.4.9). Da der Parameter p *anonym* ist, d. h. in *keiner* Aktion des Quantorrumpfs y' auftritt, ist die Menge $\Omega_a(y', p)$ der relevanten Parameterwerte der Aktion a stets leer (vgl. § 4.5.6.2). Daraus folgt, daß die Alternativen $T \in A$ eines Zustands $s = [◎, y', p, A]$ von x' nur Zustand/Wert-Paare mit dem Pseudowert p enthalten (vgl. § 4.5.7.4 und § B.2.4.4).

Wenn der Ausdruck y − und somit auch der Ausdruck y' − quasi-regulär ist, besitzt er neben seinem initialen Zustand $t_0 = \sigma(y')$ (der in jeder Alternative mit der Kardinalität ∞ vorkommt) nur $k = \eta(y') - 1$ weitere verschiedene Zustände $t_1, \ldots, t_k$. Daher kann eine Alternative T durch ein k-Tupel $[c_1, \ldots, c_k]$ von Kardinalitäten $c_i \in \mathbb{N}_0$ beschrieben werden, die angeben, wie oft der Zustand t_i (bzw. das Zustand/Wert-Paar $[t_i, p]$) in der Alternative T vorkommt.

Da jede Ausprägung eines Zustands t_i mindestens eine Aktion des Wortes w „verbraucht“ hat, kann die Kardinalität c_i eines Zustands höchstens $n = |w|$ betragen. Daher gibt es maximal n^k „gültige“ Tupel $[c_1, \ldots, c_k]$ und somit maximal n^k verschiedene Alternativen. Da jede Alternative höchstens $k+1$ Zustände enthält, kann der Verzweigungsgrad eines Zustands somit nicht größer als $(k + 1)\, n^k$ werden.

Wenn der Ausdruck y − und somit auch der Ausdruck y' − injektiv und initial fokussiert ist, wird eine Aktion a von höchstens *einem* Zustand einer Alternative akzeptiert. Aufgrund der Optimierungsfunktion ρ_1 folgt daraus, daß sich die Anzahl der Alternativen nicht vergrößern kann und somit konstant den Wert 1 behält. Außerdem ist die *Breite* einer Alternative T (d. h. die Anzahl der verschiedenen Zustand/Wert-Paare $[t, p] \in T$) höchstens $n+1$, da sie initial den Wert 1 besitzt und bei jedem Zustandsübergang höchstens um 1 zunimmt (vgl. § 4.5.7.4). Somit ist auch der Verzweigungsgrad eines Zustands höchstens von der Größenordnung n.

Wenn der Ausdruck y − und somit auch der Ausdruck y' − sowohl quasi-regulär als auch injektiv und initial fokussiert ist, lassen sich die obigen Überlegungen wie folgt kombinieren: Aufgrund der Injektivität von y' besitzt ein Zustand nur eine Alternative, deren Breite aufgrund der Quasi-Regularität von y' höchstens $k = \eta(y')$ beträgt. Somit ist auch der Verzweigungsgrad eines Zustands höchstens k, d. h. konstant.

Anmerkung: Die initiale Fokussierung des Iterationsrumpfs y wird benötigt, um zusammen mit seiner Injektivität die Injektivität des Quantorrumpfs $y' \equiv \curvearrowright y$ zu folgern (vgl. § 4.7.2.3). Die zusätzlich implizierte initiale Fokussierung von y' wird hingegen nicht benötigt.

Folgerungen

Eine parallele Iteration kann bedenkenlos angewandt werden, wenn ihr Rumpf quasi-regulär und/oder injektiv und initial fokussiert ist. Man beachte jedoch, daß diese Eigenschaften z. T. von Parameterbelegungen abhängen (vgl. § 4.7.2.3).

Aus den Überlegungen am Anfang des Beweises folgt auch, daß die Implementierung der parallelen Iteration als spezieller paralleler Quantor keine höhere Komplexität als eine direkte Implementierung besitzt, da ein Zustand $s = [\circledcirc, y', p, A]$ des Quantorausdrucks x' im wesentlichen nur eine Menge $\tilde{A}$ von Alternativen $\tilde{T}$ repräsentiert, die jeweils Multimengen von Zuständen $t \in \hat{\Theta}(y')$ darstellen. Derart strukturierte Zustände wären prinzipiell auch für eine direkte Implementierung der parallelen Iteration erforderlich.

Anmerkung

Die parallele Iteration ist nicht grundsätzlich gutartig. Als Gegenbeispiel betrachte man den Ausdruck

$$x \equiv \circledcirc\, y \qquad \text{mit} \qquad y \equiv \circledcirc\,(a - b)$$

und ein Wort $w = \langle a, \ldots, a \rangle$ der Länge $N = 1 + 2 + \ldots + n + (n{+}1) = \mathrm{O}(n^2)$.

Da es 2^n verschiedene Teilmengen $\{ k_1 \neq \ldots \neq k_m \}$ der Menge $\{ 1, \ldots, n \}$ gibt, besitzt ein solches Wort w u. a. 2^n verschiedene Zerlegungen

$$w = u_{k_0}\, u_{k_1} \ldots u_{k_m}$$

in paarweise verschiedene Teilworte u_{k_i} der Länge k_i mit $k_1, \ldots, k_m \in \{ 1, \ldots, n \}$ und $k_0 = N - \sum_{i=1}^{m} k_i > n$. Somit gibt es für den Quantorausdruck

$$x' \equiv \underset{p}{\circledcirc}\, y' \qquad \text{mit} \qquad y' \equiv \curvearrowright y \equiv \curvearrowright \circledcirc\,(a - b),$$

der zur Implementierung der parallelen Iteration x verwendet wird, mindestens 2^n verschiedene Möglichkeiten, das Wort w zu verarbeiten: Jeder oben genannten Zerlegung von w entspricht eine Alternative

$$T = \sum_{1}^{\infty} \big[\sigma(y'),\, p\big] + \sum_{i=0}^{m} \big[t_{k_i},\, p\big]$$

des Zustands $\hat{\sigma}_w(x')$ mit Teilzuständen

$$t_k = \hat{\sigma}_{u_k}(y') \qquad \text{für } k \in \{ k_0, \ldots, k_m \}.$$

Da der Ausdruck $y' \equiv \curvearrowright \circledcirc\,(a - b) = \circledcirc\,(a - b)$ nach Verarbeitung eines Wortes $u_k = \langle a, \ldots, a \rangle$ der Länge k genau k b's akzeptiert, müssen die Zustände t_k für verschiedene Werte von k offensichtlich verschieden sein. Daraus folgt jedoch, daß auch die oben genannten 2^n Alternativen T paarweise verschieden sind.

Somit besitzt der Zustand $\hat{\sigma}_w(x')$ des Quantorausdrucks x' mindestens den Verzweigungsgrad 2^n, obwohl die Länge des Wortes w nur von der Größenordnung n^2 ist.

Dieses Beispiel widerlegt die über längere Zeit gehegte Hoffnung, *elementare Interaktionsausdrücke* seien möglicherweise grundsätzlich gutartig. Es zeigt außerdem, daß durch die Hinzunahme der parallelen Iteration ein gewaltiger *Komplexitätssprung* eintritt: Elementare Ausdrücke *ohne* parallele Itera-

tion, d. h. quasi-reguläre Ausdrücke, sind *harmlos*, während elementare Ausdrücke *mit* paralleler Iteration potentiell *bösartig* sind.

4.7.4 Aussagen zu Quantoren

4.7.4.1 Boolesche Quantoren

Satz

1. Boolesche Quantoren (Disjunktions-, Konjunktions- und Synchronisations-Quantoren) sind gutartig.
2. Ist der Ausdruck $x \equiv \underset{p}{\bigcirc}\, y$ vollständig quantifiziert, sind Disjunktions- und Konjunktions-Quantoren sogar harmlos.

Beweis

1. Der Verzweigungsgrad eines Zustands $s = [\bigcirc, y, p, r, T]$ entspricht im wesentlichen der Kardinalität der Menge T, die initial den Wert 0 besitzt ($T = \emptyset$) und bei jedem Zustandsübergang höchstens um eine Konstante $k = \max\limits_{a \in \alpha(y)} |a|$ zunimmt (vgl. § 4.5.6.2). Somit beträgt die Kardinalität von T höchstens $k\,n$ mit $n = |w|$.
2. Wenn der Ausdruck x vollständig quantifiziert ist, so ergibt sich ähnlich wie in § B.4.6:

$$\hat{\sigma}_w(x) = \begin{cases} [\bigcirc, y, p, \sigma(y), \emptyset] & \text{für } w = \langle\rangle, \\ [\bigcirc, y, p, \bot, T] \quad \text{mit} \quad T = \left\{ \left[\hat{\sigma}_w\left(y_p^{\omega}\right), \omega\right] \,\middle|\, \omega \in \Omega' \right\} & \text{für } w = \langle a, \ldots \rangle \end{cases}$$

   mit einer Menge $\Omega' \subseteq \Omega_a(y, p)$. Daher ist die Kardinalität der Menge T in diesem Fall durch die Konstante k beschränkt.

Folgerungen

Boolesche Quantoren können grundsätzlich bedenkenlos eingesetzt werden. Im „Normalfall", d. h. bei vollständig quantifizierten Rümpfen, sind Disjunktions- und Konjunktions-Quantoren sogar harmlos.

Anmerkungen

1. Boolesche Quantoren sind nicht grundsätzlich harmlos. Als Gegenbeispiel betrachte man den Ausdruck

$$x \equiv \underset{p}{\bigcirc} \left(a(p) \circledcirc \underset{q}{\circledcirc}\, a(q) \right)$$

   und ein Wort $w = \langle a(\omega_1), a(\omega_2), \ldots \rangle$ mit paarweise verschiedenen Werten $\omega_1 \neq \omega_2 \neq \ldots \in \Omega$. In diesem Fall wächst der Verzweigungsgrad der Zustände $\hat{\sigma}_w(x)$ tatsächlich linear mit der Länge des Wortes w.
2. Auch vollständig quantifizierte Synchronisations-Quantoren sind normalerweise nicht harmlos, wie das einfache Gegenbeispiel $x = \underset{p}{\bigcirc}\, a(p)$ zeigt. Anders als bei Disjunktions- und Konjunktions-Quantorausdrücken, bleibt der Zustand des abstrakten Zweigs y hier bei jedem Zustandsübergang unverändert $\sigma(y)$, da für jede Aktion $a \in \Sigma$ gilt: $a \notin \alpha(y)$. Für ein Wort w wie oben wächst daher auch hier der Verzweigungsgrad der Zustände $\hat{\sigma}_w(x)$ linear bzgl. der Länge von w.

4.7.4.2 Parallele Quantoren

Satz

Parallele Quantoren sind gutartig, wenn ihr Rumpf vollständig und homogen quantifiziert ist.

Beweis

Aus der vollständigen und homogenen Quantifizierung eines Ausdrucks $x \equiv \underset{p}{\circledcirc} y$ folgt gemäß § 4.7.2.2, daß die Alphabete $\alpha\left(y_p^\omega\right)$ für $\omega \in \Omega$ paarweise disjunkt sind. Folglich kann jede Aktion a eines Wortes $w \in \Psi(x)$ *eindeutig* einem Zweig y_p^ω des Quantorausdrucks x zugeordnet werden, d. h. es gibt eine *eindeutige* Zerlegung des Wortes w in Teilworte $u_1, \ldots, u_k$ mit zugehörigen Werten $\omega_1, \ldots, \omega_k \in \Omega$, für die

$$w \in \bigotimes_{i=1}^{k} u_i \qquad \text{und} \qquad u_i \in \Psi\left(y_p^{\omega_i}\right) \qquad \text{für } i = 1, \ldots, k$$

gilt. Aufgrund der Optimierung

$$\rho_1(s) = \left[\circledcirc, y, p, \hat{A}\right] \qquad \text{mit} \qquad \hat{A} = \left\{ T \in A \,\middle|\, \bigwedge_{t \in T} \psi(t) \right\},$$

bei der Alternativen T mit ungültigen Teilzuständen t entfernt werden, besteht die Menge $\hat{A}$ eines optimierten Zustands $\hat{s} = \rho_1(s)$ daher aus einer einzigen Alternative.

Die Breite einer Alternative T, d. h. die Anzahl der verschiedenen Zustand/Wert-Paare $[t, \omega] \in T$, ist auch hier (ähnlich wie beim Spezialfall der parallelen Iteration in § 4.7.3.4) von der Größenordnung n, da sie initial den Wert 1 besitzt und bei jedem Zustandsübergang höchstens um einen konstanten Wert zunimmt. Somit ist der Verzweigungsgrad der Zustände $s = [\circledcirc, y, p, A]$ ebenfalls von der Größenordnung n.

Folgerung

Da die vollständige und homogene Quantifizierung eines Ausdrucks − obwohl sie auf den ersten Blick restriktiv erscheinen mag − in praktischen Anwendungen des „Allquantors" den Normalfall darstellt, kann dieser Operator normalerweise bedenkenlos angewandt werden.

Anmerkungen

1. Parallele Quantorausdrücke, die lediglich vollständig, nicht jedoch homogen quantifiziert sind, sind nicht notwendigerweise gutartig. Als Gegenbeispiel betrachte man den Ausdruck

   $$x \equiv \underset{p}{\circledcirc} y \qquad \text{mit} \qquad y \equiv \left(\underset{q}{\bigcirc} a(p, q)\right) \circ \left(\underset{q}{\bigcirc} a(q, p)\right)$$

   und ein Wort

   $$w = \langle a(\omega_1, \omega_2), a(\omega_3, \omega_4), \ldots \rangle$$

   mit paarweise verschiedenen Werten $\omega_1 \neq \omega_2 \neq \ldots \in \Omega$. Da der Ausdruck nicht homogen quantifiziert ist, sind die Alphabete der Zweige y_p^ω für verschiedene Werte $\omega \in \Omega$ *nicht* disjunkt, sondern es gilt:

   $$\alpha\left(y_p^{\omega_i}\right) \cap \alpha\left(y_p^{\omega_j}\right) = \left\{ a(\omega_i, \omega_j), a(\omega_j, \omega_i) \right\}.$$

   Daher wird eine Aktion $a(\omega_i, \omega_j)$ stets von zwei Zweigen $y_p^{\omega_i}$ und $y_p^{\omega_j}$ des Ausdrucks x akzeptiert, was zur Folge hat, daß sich die Kardinalität der Menge A bei jedem Zustandsübergang verdoppelt.

2. Ebenso sind parallele Quantorausdrücke, deren Rumpf quasi-regulär ist, nicht notwendigerweise gutartig. Als Gegenbeispiel betrachte man den Ausdruck

$$x \equiv \underset{p}{\circledcirc} \, (a(p) - b)$$

und ein Wort

$$w = \langle a(\omega_1), a(\omega_2), b, a(\omega_3), a(\omega_4), b, \ldots \rangle$$

mit paarweise verschiedenen Werten $\omega_1 \neq \omega_2 \neq \ldots \in \Omega$. Hier wird die Kardinalität der Menge A durch jedes b mindestens verdoppelt.

4.7.5 Zusammenfassung

4.7.5.1 Komplexitätshierarchie von Interaktionsausdrücken

Aus den Resultaten der vorangegangenen Abschnitte läßt sich die folgende *Komplexitätshierarchie* von Interaktionsausdrücken ableiten (vgl. Abb. 4.26):

1. *Quasi-reguläre Ausdrücke*, d. h. Ausdrücke, die nur die unären Operatoren $\smile$ und $\ominus$, die Booleschen Operatoren $\circ$, $\boldsymbol{\circ}$ und $\bullet$ sowie die Kompositionsoperatoren $-$ und $\circledcirc$ enthalten, sind *harmlos* (vgl. § 4.7.3.1 bis § 4.7.3.3).
2. Nimmt man die *Booleschen Quantoren* $\underset{p}{\bigcirc}$, $\underset{p}{\boldsymbol{\bigcirc}}$ und $\underset{p}{\CIRCLE}$ hinzu, so sind die resultierenden Ausdrücke *gutartig*, sofern man die potentiell bösartige parallele Komposition ausschließt (vgl. § 4.7.3.1, § 4.7.3.2 und § 4.7.4.1).
3. Durch die Hinzunahme der *parallelen Operatoren* $\circledcirc$, $\circledcirc\!\!\circ$ und $\underset{p}{\circledcirc}$ erhält man *potentiell bösartige* Ausdrücke (vgl. § 4.7.3.3 und § 4.7.3.4).

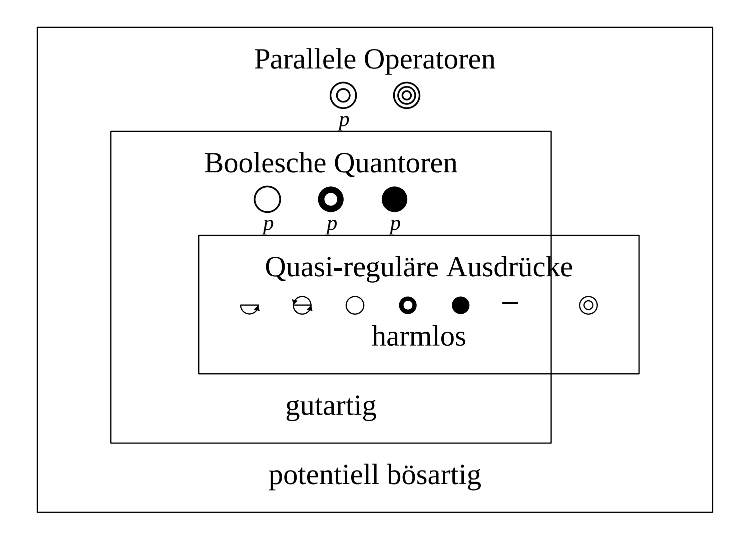


Abbildung 4.26: Komplexitätshierarchie von Interaktionsausdrücken

Anmerkung: Bereits die Kombination von paralleler Komposition und Booleschen Quantoren ist potentiell bösartig, wie das Beispiel $x \equiv y \circledcirc y$ mit $y \equiv \underset{p}{\boldsymbol{\bigcirc}} a(p)$ zeigt. Für ein Wort $w = \langle a(\omega_1), \ldots, a(\omega_n) \rangle$ mit paarweise verschiedenen Werten $\omega_1 \neq \ldots \neq \omega_n \in \Omega$ wächst der Zustands-

baum des Ausdrucks x in derselben Weise wie der des bösartigen Ausdrucks $x' \equiv y' \circledcirc y'$ mit $y' \equiv \underset{p}{\circledcirc} a(p)$ (vgl. § 4.7.3.3).

4.7.5.2 Tabellarischer Überblick

Tabelle 4.27 faßt die Resultate der vorangegangenen Abschnitte detaillierter zusammen, indem für jeden Operator von Interaktionsausdrücken hinreichende Bedingungen für seine Harmlosigkeit bzw. Gutartigkeit und − sofern der Operator potentiell bösartig ist − ein Beispiel eines bösartigen Ausdrucks genannt werden. Die Tabelle kann daher als Checkliste dienen, mit deren Hilfe für einen gegebenen Ausdruck überprüft werden kann, ob er harmlos oder gutartig ist.

Operator	Hinreichende Bed. für *Harmlosigkeit*	Hinreichende Bed. für *Gutartigkeit*	Beispiel für *Bösartigkeit*	*Abschnitt*
$\circlearrowright$, $\bigcirc$, $\odot$, $\bullet$	$\top$	$\top$		4.7.3.1
$y - z$	$y \in$ TF oder $z \in$ QR oder $\alpha(y) \cap \alpha(z) = \emptyset$	$\top$		4.7.3.2
$\ominus\, y$	$y \in$ TF $\cup$ QR	$\top$		4.7.3.2
$y \circledcirc z$	$y, z \in$ QR oder $\alpha(y) \cap \alpha(z) = \emptyset$	?	$\left(\underset{p}{\circledcirc} a(p)\right) \circledcirc \left(\underset{p}{\circledcirc} a(p)\right)$	4.7.3.3
$\circledcirc\!\!\circ\, y$	$y \in$ QR $\cap$ IN $\cap$ IF	$y \in$ QR $\cup$ (IN $\cap$ IF)	$\circledcirc\!\!\circ\, \circledcirc\!\!\circ\, (a - b)$	4.7.3.4
$\underset{p}{\bigcirc} y$, $\underset{p}{\bullet}\, y$	$y \in$ VQ	$\top$		4.7.4.1
$\underset{p}{\odot} y$	?	$\top$		4.7.4.1
$\underset{p}{\circledcirc}\, y$	?	$y \in$ VQ $\cap$ HQ	$\underset{p}{\circledcirc}\, (a(p) - b)$	4.7.4.2
$\overset{n}{\circledcirc}\, y$	$y \in$ QR $\cup$ IN	?	$\overset{2}{\circledcirc}\, \underset{p}{\circledcirc}\, a(p)$	

Verwendete Abkürzungen		
Menge	*Eigenschaft*	*Abschnitt*
QR	quasi-regulär	4.7.2.1
VQ HQ	vollständig quantifiziert homogen quantifiziert	4.7.2.2
IN IF TF	injektiv initial fokussiert terminal fokussiert	4.7.2.3

Tabelle 4.27: Komplexität der einzelnen Operatoren

Die Bedingung $\top$ besagt, daß der Operator grundsätzlich harmlos bzw. gutartig ist, während ein Fragezeichen anzeigt, daß kein entsprechendes Kriterium bekannt ist bzw. in den vorangegangenen Abschnitten vorgestellt wurde. Da die operationale Semantik und Implementierung von Multiplikatorausdrücken nicht weiter betrachtet wurde, werden die Aussagen für $\circledcirc$-Multiplikatoren ohne Beweis genannt. Ebenso werden die Beispiele für bösartige Ausdrücke nicht näher erläutert.

4.7.5.3 Beispiele

Die folgenden Beispiele zeigen exemplarisch, wie sich die Gutartigkeit eines gegebenen Ausdrucks schrittweise mit Hilfe von Tab. 4.27 herleiten läßt. Die in der Tabelle angegebenen Abschnittsnummern erleichtern im Zweifelsfall das Nachschlagen der entsprechenden Sätze.

Beispiel 1

Der Ausdruck

$$\underset{p}{\circledcirc} \ominus \Bigg\{ \Bigg(\text{⊚} \underset{u}{\bigcirc} \mathsf{vorbereiten}(p, u) \Bigg) \circ \Bigg(\text{⊚} \underset{u}{\bigcirc} \mathsf{aufklären}(p, u) \Bigg) \circ \\ \Bigg(\underset{u}{\bigcirc} (\mathsf{abrufen}(p, u) - \mathsf{untersuchen}(p, u)) \Bigg) \Bigg\},$$

der im Beispiel der medizinischen Untersuchungsworkflows die Integritätsbedingung für Patienten spezifiziert (vgl. § 2.7.1.2), ist gutartig, denn:

1. Die sequentielle Komposition $\mathsf{abrufen}(p, u) - \mathsf{untersuchen}(p, u)$ ist regulär und daher für jede Belegung der Parameter p und u *harmlos*.
2. Die drei Disjunktionsquantoren $\underset{u}{\bigcirc}$ sind für jede Belegung von p *harmlos*, da sie vollständig quantifiziert sind.
3. Die Disjunktions-Quantorausdrücke sind außerdem für jede Belegung von p injektiv und initial fokussiert, da ihre Rümpfe für jede Belegung von p und u diese Eigenschaften besitzen und die Ausdrücke vollständig und homogen quantifiziert sind.
   Folglich sind die beiden parallelen Iterationen für jede Belegung von p *gutartig*.
4. Die disjunktive Verknüpfung der drei Teilausdrücke ist grundsätzlich *harmlos*.
5. Die sequentielle Iteration ist grundsätzlich *gutartig*.
6. Die Allquantifizierung $\underset{p}{\circledcirc}$ ist *gutartig*, da ihr Rumpf vollständig und homogen quantifiziert ist.

Anmerkung: Betrachtet man die Gutartigkeits-Beweise in den Abschnitten 4.7.3.4, 4.7.3.2 und 4.7.4.2 genauer, so erkennt man, daß der Verzweigungsgrad der Operatoren ⊚, $\ominus$ und $\underset{p}{\circledcirc}$ höchstens *linear* bzgl. der Länge n der verarbeiteten Aktionsfolge wächst. Die baumartige Repräsentation eines Zustands des Gesamtausdrucks besitzt daher auf maximal drei Ebenen jeweils einen oder zwei Knoten mit linearem Verzweigungsgrad, während alle übrigen Knoten konstanten Verzweigungsgrad besitzen. Somit ist die Gesamtgröße eines Zustands höchstens von der Größenordnung n^3.

Die Formulierung *höchstens* soll hierbei zum Ausdruck bringen, daß die tatsächlichen Verzweigungsgrade auch noch stark von der konkret ausgeführten Aktionsfolge abhängen. Beispielsweise wächst der Verzweigungsgrad der beiden parallelen Iterationen nur durch Ausführung einer Startaktion[32] $\mathsf{vorbereiten}_0$ bzw. $\mathsf{aufklären}_0$, während er durch die Ausführung einer entsprechenden Endeaktion sogar wieder schrumpft![33] Für realistische Aktionsfolgen, bei denen Start- und Endeaktionen von Aktivitäten nicht beliebig weit auseinanderliegen, ist der Verzweigungsgrad der parallelen Iterationen daher sogar konstant.

Auch der Verzweigungsgrad des parallelen Quantorzustands wächst nur dann wirklich linear, wenn die ausgeführten Aktionen immer wieder neue Werte des Parameters p enthalten. Wenn eine Aktionsfolge der Länge n jedoch Aktionen mit $O(n)$ verschiedenen Werten für p enthält, so kann sie nur noch $O(1)$ Aktionen pro Wert enthalten. In diesem Fall hätten daher alle Teilzustände des parallelen Quan-

[32] Beachte: Bei allen Tätigkeiten des Ausdrucks handelt es sich um *Aktivitäten*, d. h. um Sequenzen einer Start- und einer Endeaktion (vgl. § 2.6.2).

[33] Durch die Ausführung einer solchen Aktion geht der terminal fokussierte Disjunktions-Quantorausdruck $\underset{u}{\bigcirc} \dots$ in seinen Endzustand $\top$ über. Dadurch reduziert sich die Breite der im Beweis von § 4.7.3.4 betrachteten Alternative T des Zustands der parallelen Iteration um 1, sofern T diesen Zustand bereits ein- oder mehrmals enthält (was nach Ausführung der ersten derartigen Aktion der Fall ist).

tors konstante Größe (weil sie nur O(1) Aktionen verarbeitet haben) und der Gesamtzustand daher nur die Größenordnung n. Dieses Beispiel zeigt, daß die Verzweigungsgrade verschachtelter Operatoren in vielen Fällen nicht unabhängig voneinander sind und ihr Produkt daher nur eine grobe obere Schranke darstellt.

Beispiel 2

Auch der Ausdruck

$$\underset{u}{\circledcirc}\ \overset{5}{\circledcirc}\ \circlearrowright\ \underset{p}{\bigcirc}\ \mathsf{untersuchen}(p, u),$$

der die allgemeine Kapazitätsbeschränkung für Untersuchungsstellen beschreibt (vgl. § 2.7.2.1), ist gutartig:

1. Der Disjunktionsquantor $\underset{p}{\bigcirc}$ ist für alle Belegungen des Parameters u *harmlos*, da sein Rumpf vollständig quantifiziert ist.
   Außerdem ist er für alle Werte von u injektiv und beidseitig fokussiert, da sein Rumpf für alle Werte von u und p diese Eigenschaften besitzt.

2. Die sequentielle Iteration $\circlearrowright$ ist für alle Werte von u *harmlos*, da ihr Rumpf jeweils terminal fokussiert ist.
   Außerdem ist sie für alle Werte von u injektiv, da ihr Rumpf jeweils injektiv und beidseitig fokussiert ist.

3. Der parallele Multiplikator $\overset{5}{\circledcirc}$ ist für alle Werte von u *harmlos*, da sein Rumpf jeweils injektiv ist.

4. Der parallele Quantor $\underset{p}{\circledcirc}$ ist *gutartig*, da sein Rumpf vollständig und homogen quantifiziert ist.

Anmerkung: Auch hier gilt, wie in Beispiel 1, daß der Verzweigungsgrad des parallelen Quantorzustands höchstens linear bzgl. der Länge n der ausgeführten Aktionsfolge wächst. Somit ist die Größe der Zustände des Gesamtausdrucks in diesem Beispiel höchstens von der Größenordnung n.

Beispiel 3

Der Ausdruck

$$\underset{k}{\circledcirc}\ \circledcirc\ \{\ \mathsf{10\,Pf}(k) - [\mathsf{Kopie\,A4}(k) \circ (\mathsf{10\,Pf}(k) - \mathsf{Kopie\,A3}(k))]\ \},$$

der ein Kopiergeschäft mit beliebig vielen A4/A3-Kopierern beschreibt (vgl. § 2.3.4.3), ist ebenfalls gutartig:

1. Der gesamte Rumpf der parallelen Iteration ist regulär und daher für jede Belegung des Parameters k *harmlos*.

2. Die parallele Iteration $\circledcirc$ ist für jeden Wert von k *gutartig*, da ihr Rumpf (quasi-)regulär ist.

3. Der Allquantor $\underset{k}{\circledcirc}$ ist *gutartig*, da sein Rumpf vollständig und homogen quantifiziert ist.

Anmerkung: Da der Rumpf $y \equiv \{\ \mathsf{10\,Pf}(k) - \dots\ \}$ der parallelen Iteration in diesem Beispiel *nicht* injektiv ist (er akzeptiert in zwei verschiedenen Zuständen die Aktion $\mathsf{10\,Pf}(k)$), ist der Verzweigungsgrad der parallelen Iteration hier nicht notwendig linear beschränkt. Stattdessen kann man dem Beweis in § 4.7.3.4 entnehmen, daß der Verzweigungsgrad die Größenordnung n^k mit $k = \eta(y') - 1$ und $y' \equiv \circlearrowright y$ besitzt. Da die Zustandszahl $\eta(y')$ offensichtlich gleich 4 ist (ein Läufer kann sich am Anfang des entsprechenden Graphen, am Verzweigungsknoten der Entweder-oder-Verzweigung, zwischen den Aktionen $\mathsf{10\,Pf}(k)$ und $\mathsf{Kopie\,A3}(k)$ des unteren Entweder-oder-Zweigs oder am Ende des Graphen befinden), erhält man konkret die Größenordnung n^3.

Da der Verzweigungsgrad des parallelen Quantorzustands wieder maximal lineare Größe besitzt, erhält man insgesamt Zustände der Größenordnung $\leq n^4$. Da aber auch in diesem Beispiel die Verzweigungsgrade der Operatoren $\underset{k}{\circledcirc}$ und $\circledcirc$ vermutlich nicht unabhängig voneinander sind, ist auch diese Abschätzung eher pessimistisch.

4.7.5.4 Resümee

Ebenso wie für diese drei „repräsentativen" Beispiele läßt sich mit Hilfe der vorgestellten Komplexitätsaussagen für *sämtliche* in dieser Arbeit genannten Beispiele[34] − insbesondere für die des Abschnitts 2.7 − zeigen, daß sie gutartig oder sogar harmlos sind. Diese Tatsache − zusammen mit der Erfahrung anderer praktischer Beispiele − gibt Grund zu der Annahme, daß die meisten (wenn nicht alle) praktisch relevanten Ausdrücke gutartig sind und daß bösartige Ausdrücke − zusammen mit einem „passenden" Wort − gezielt konstruiert werden müssen.

4.8 Rückblick

Nach diesen grundsätzlich positiven und erfreulichen Aussagen zur Komplexität der entwickelten Implementierung, sollen im nachfolgenden Abschnitt 4.8.1 die wesentlichen Faktoren zusammengefaßt werden, die für diesen Erfolg ausschlaggebend sind. Außerdem wird in § 4.8.2 kurz auf den Umfang und die Qualität der entwickelten Software eingegangen.

4.8.1 Erfolgsfaktoren

4.8.1.1 Operationale Semantik

Bereits in § 4.4.4 wurde erläutert, daß sich die in Kapitel 3 entwickelte *formale Semantik* von Interaktionsausdrücken zwar *prinzipiell*, aber nicht *effizient* implementierungstechnisch umsetzen läßt. Betrachtet man den dort vorgestellten Algorithmus noch einmal genauer, so erkennt man, daß er grundsätzlich weder in der Lage ist, spezielle Eigenschaften des gegebenen *Ausdrucks* (wie z. B. Quasi-Regularität, Injektivität o. ä.) noch den konkreten Aufbau des vorliegenden *Wortes* gewinnbringend auszunutzen.

Aus diesem Grund stellt die Entwicklung einer separaten *operationalen Semantik* einen ersten wichtigen Schritt in Richtung einer effizienten Implementierung dar, weil ein geeignet konstruiertes und optimiertes Zustandsmodell prinzipiell in der Lage ist, sowohl vorteilhafte Eigenschaften eines Ausdrucks als auch den konkreten Aufbau eines Wortes auszunutzen. So bleibt die Größe paralleler Kompositions-Zustände beispielsweise in vielen Fällen konstant, obwohl der Operator *potentiell* bösartig ist, und selbst die in Tab. 4.27 (§ 4.7.5.2) genannten bösartigen Ausdrücke verhalten sich für sehr viele konkrete *Worte* immer noch gutartig!

4.8.1.2 Teilwortorientierte Semantik

Neben den in § 3.2.3 genannten Gründen, stellt die *effiziente Implementierbarkeit* ein weiteres wichtiges Argument für die Verwendung einer *teilwortorientierten Semantik*, d. h. einer *unabhängigen* Definition der Menge der partiellen Worte, dar − zumindest was die Lösung des *Aktionsproblems* betrifft. Würde man nämlich, dem „klassischen" Ansatz folgend, verlangen, daß ein Ausdruck mit Sackgassen (d. h. ein insgesamt unerfüllbarer Ausdruck) bereits im ersten Schritt keine Aktion akzeptieren darf, so wäre das operationale Modell in der vorliegenden Form zum Scheitern verurteilt, weil es im allgemeinen erst nach einigen Zustandsübergängen „erkennen" kann, daß ein Ausdruck eine Sackgasse, d. h. einen unerfüllbaren Teilausdruck besitzt.

[34] mit Ausnahme der expliziten Negativbeispiele

Sofern man nur an einer Lösung des *Wortproblems* interessiert ist, stellt dies (ebenso wie bei endlichen Automaten) kein Problem dar, weil man sich in diesem Fall nur für den Wert des Prädikats $\varphi(s)$ des *letzten* Zustands $s = \sigma_w(x)$ interessiert. Zur Lösung des Aktionsproblems muß man jedoch anhand eines *Zwischenzustands* $\sigma_u(x)$ mit einem Präfix $u \neq w$ entscheiden, ob die nächste vorliegende Aktion akzeptiert werden darf oder nicht. Wollte man hier die klassische Semantik implementieren, die eine Aktion nur akzeptiert, wenn das bis jetzt verarbeitete Wort zu einem *vollständigen* Wort des Ausdrucks *ergänzt* werden kann, so bräuchte man zusätzliche (theoretische) Methoden und (praktische) Algorithmen zur Beantwortung dieser Frage − wobei zuallererst zu klären wäre, ob sie überhaupt entscheidbar ist (vgl. auch § 2.5.1.2).

4.8.1.3 Sortierte Sequenzen und Vergleichsprozeduren

In § 2.4.4 wurde anhand eines Beispielausdrucks bzw. -graphen erläutert, daß die Zustände einer parallelen Iteration bei einer „naiven" Vorgehensweise exponentielle Größe (bzgl. der Länge des Wortes w) besitzen würden, obwohl der Ausdruck gemäß § 4.7.5.3 (Beispiel 3) gutartig ist. Die entscheidende Beobachtung dort, daß Alternativen eines Zustands *äquivalent* sind, wenn sie durch Vertauschen von Tupelkomponenten (d. h. von Teilzuständen einer Alternative) ineinander übergeführt werden können, wird im mathematischen Modell einfach dadurch umgesetzt, daß Alternativen als *Multimengen* repräsentiert werden, bei denen lediglich die *Kardinalität*, nicht jedoch die *Anordnung* der Elemente relevant ist. Die einzelnen Alternativen wiederum stellen − ebenso wie z. B. die Teilzustände einer sequentiellen Komposition oder Iteration − Elemente einer *Menge*, d. h. eines *duplikatfreien* „Containers" dar.

Das mathematische Modell würde sich nur unwesentlich ändern, wenn man anstelle von Mengen oder Multimengen jeweils geordnete *Tupel* verwenden würde, bei denen die Anordnung ihrer Elemente *relevant* ist. Da sich derartige Container − insbesondere in einer traditionellen imperativen Programmiersprache wie z. B. C − implementierungstechnisch wesentlich leichter umsetzen lassen als Mengen − beispielsweise als *unsortierte Listen* oder *dynamische Arrays*, bei denen Elemente grundsätzlich am Anfang oder am Ende eingefügt werden −, wurden Alternativen eines parallelen Quantorzustands in einer ersten prototypischen Implementierung tatsächlich in dieser Form repräsentiert. Diese scheinbar unwesentliche Entwurfsentscheidung führte dazu, daß die Laufzeit des Programms für bestimmte Ausdrücke „unerklärlich" groß war.

Erst durch eine sorgfältige Analyse der resultierenden Zustände − zusammen mit der Überzeugung, daß sich Ausdrücke wie der in § 2.4.4 betrachtete intuitiv wesentlich effizienter verarbeiten lassen müßten − ergab sich, daß man *äquivalente* Zustände am einfachsten dadurch erkennen und eliminieren kann, daß man Mengen als *sortierte* (und duplikatfreie) Sequenzen implementiert. Bei dieser Vorgehensweise sind zwei Mengen nämlich genau dann im mathematischen Sinne *gleich*, wenn sie als Sequenzen *identisch* sind, d. h. die gleichen Elemente in *derselben* Reihenfolge enthalten. Ein positiver Nebeneffekt dieser Vorgehensweise besteht in der einfachen und effizienten Realisierung sämtlicher Mengenoperationen (wie z. B. Elementtest durch *binäre Suche* mit logarithmischer Komplexität oder Vereinigung und Durchschnittsbildung mit Hilfe von *Sort-merge-Algorithmen* mit linearer Komplexität) ohne zusätzlichen Verwaltungsaufwand (wie er z. B. bei der Verwendung von Hashtabellen anfallen würde).

Eine wesentliche Voraussetzung für die Realisierung sortierter Sequenzen ist das in § 4.4.2.3 erwähnte Konzept der *Vergleichsprozeduren* für benutzerdefinierte Typen (wie z. B. Parameter, Aktionen und Zustände), mit denen − auf einfache Art und Weise − eine *totale Ordnung* auf den Objekten eines Typs definiert wird. Daß diese Ordnung möglicherweise „künstlich" ist, weil es für die betrachteten Objekte (wie z. B. Mengen oder Multimengen) in der „realen Welt" entweder nur eine *partielle* oder aber gar keine „natürliche" Ordnung gibt, spielt hierbei keine Rolle; sobald man Objekte elektronisch repräsentiert und verarbeitet, kann man prinzipiell eine totale Ordnung für sie definieren, notfalls als lexikographische Ordnung von Bitfolgen.

4.8.1.4 Zustandsoptimierungen

Obwohl durch die Verwendung sortierter Sequenzen bereits ein großes Optimierungspotential des Zustandsmodells ausgeschöpft ist, können zusätzliche *Zustandsoptimierungen* zu einer weiteren − teils geringfügigen, teils auch wesentlichen − Verbesserung der Effizienz der Implementierung beitragen. Der entscheidende Faktor ist hierbei in der Regel *nicht*, daß ein „großer“ Zustand durch einen „kleineren“ ersetzt wird, sondern daß Zustände als gleich oder *äquivalent* erkannt werden können, die ansonsten sowohl mathematisch als auch implementierungstechnisch *verschieden* sind.

Beispielsweise beruht die Injektivität des Ausdrucks $y \equiv \ominus \underset{p}{\bigcirc} a(p)$ entscheidend auf der in § 4.5.7.1 genannten Optimierung $\rho_2\big([\circ, y, p, \bot, \{ [\top, \omega] \}]\big) = \top$, ohne die der Disjunktionsquantorausdruck $\underset{p}{\bigcirc} a(p)$ nicht terminal fokussiert und die Iteration demnach nicht injektiv wäre. *Ohne* diese Optimierung würde sich die parallele Iteration $x \equiv \circledcirc\, y$, die gemäß § 4.7.3.4 gutartig ist, für ein Wort $w = \langle a\big(\omega_1\big), \ldots, a\big(\omega_n\big)\rangle$ (mit paarweise verschiedenen Werten $\omega_1 \neq \ldots \neq \omega_n$) tatsächlich *bösartig* verhalten!

4.8.1.5 Fazit

Aus den vorangegangenen Erläuterungen folgt zum einen, daß die Effizienz der entwickelten Implementierung keineswegs selbstverständlich ist, sondern nur durch das Zusammenwirken einer Reihe wichtiger Faktoren zustande kommt. Sowohl das „Aufspüren“ dieser Faktoren als auch ihre theoretische und praktische Umsetzung war z. T. mit erheblichem Aufwand verbunden.

Zum anderen legen diese Erkenntnisse die Vermutung nahe, daß das vorhandene Optimierungspotential noch nicht vollständig ausgeschöpft ist, d. h. daß es möglicherweise weitere, bisher nicht berücksichtigte Faktoren gibt, mit denen sich die Effizienz der Implementierung noch weiter verbessern läßt. Beispielsweise wurde in § 4.7.3.3 angedeutet, daß sich bösartige Ausdrücke mit Hilfe von *Äquivalenztransformationen* u. U. in gutartige verwandeln lassen.

Eine weitere Verbesserungsmöglichkeit besteht in der Verwendung eines „Zustandsübergangs-Caches“, in dem für jeden Teilzustand eines parallelen Quantorzustands, für den bereits ein Zustandsübergang (mit einer bestimmten Belegung des Quantorparameters) durchgeführt wurde, der zugehörige Folgezustand zwischengespeichert wird. Da verschiedene Alternativen eines parallelen Quantorzustands häufig gemeinsame Teilzustände besitzen, kann man auf diese Weise u. U. zahlreiche Zustandsübergangs-Berechnungen einsparen. Interessant ist, daß das `CH`-Konzept der offenen Typen eine sehr einfache und effiziente Realisierung eines solchen Zwischenspeichers erlaubt, indem der Typ `State` lediglich um ein Attribut `cache` (ebenfalls vom Type `State`) erweitert wird, in dem jeweils der Folgezustand eines Teilzustands hinterlegt werden kann. (Am Ende eines gesamten Zustandsübergangs des Quantorzustands müssen diese „Mini-Caches“ jedoch wieder gelöscht werden.)

4.8.2 Umfang und Qualität der Implementierung

Zum Abschluß dieses umfangreichen Kapitels gibt Tab. 4.28 einen groben Überblick über den Gesamtumfang der entwickelten Software. Betrachtet man die genannten Zahlen etwas genauer, so wird deutlich, daß etwa drei Fünftel des Programmcodes (1630 Zeilen) in Form wiederverwendbarer *Bibliotheksfunktionen* (vorwiegend C++-Template-Funktionen) extrahiert werden konnte, so daß der eigentliche Anwendungscode (1070 Zeilen) kaum unnötigen, die Lesbarkeit und Verständlichkeit behindernden „Ballast“ enthält. Insbesondere muß man sich an keiner Stelle des Programms um die Allokation oder Deallokation von Speicherplatz kümmern, obwohl die verwendeten Datenstrukturen hochgradig dynamisch sind. Dementsprechend weist der Code über weite Strecken einen fast „funktionalen“ Charakter auf.

Die im ersten Teil der Tabelle erwähnten Präprozessoren `import` und `CHpre` werden in Anhang A (§ A.3.1) etwas genauer erläutert.

Programm/ Modul	Funktionalität	Sprache	Umfang (Codezeilen)
	Präprozessoren		*700*
`import`	Realisierung des CH-Modulkonzepts	AWK	430
`CHpre`	Realisierung der übrigen CH-Spracherweiterungen	AWK	270
	CH-*Bibliothek*		*1630*
`seq`	Dynamische Sequenzen, Mengen und Multimengen	C++	860
`str`	Dynamische Zeichenketten (Strings)	C++	150
`obj`	Offene Typen und zugehörige Objekte	C++	530
`cmp`	Vergleichsprozeduren und -operatoren	C++	50
`part`	Partielle Funktionen	C++	40
	Implementierung von Interaktionsausdrücken		*1070*
`expr`	Operatorbäume	CH	150
`parse`	Parser für Interaktionsausdrücke	CH, Yacc	230
`state`	Zustandsmodell	CH	640
`main`	Hauptprogramm	CH	50
	Gesamtsumme		*3400*

Tabelle 4.28: Umfang der Implementierung

Rückblickend betrachtet, hat sich die sorgfältige Entwicklung des mathematischen Zustandsmodells (die im Verlauf der Arbeit erst *nach* der Entwicklung einiger prototypischer Implementierungen erfolgte) in verschiedener Hinsicht positiv auf die Qualität des Programms ausgewirkt. Zum einen wurden durch die Verifikation des Zustandsmodells einige subtile Fehler (im Sinne fehlerhafter Grundannahmen) in früheren Programmversionen, insbesondere im Kontext von Quantoren, aufgedeckt. Beispielsweise wurde über lange Zeit die intuitiv durchaus einleuchtend erscheinende Aussage für wahr gehalten, daß ein konkreter Zweig y_p^ω eines Quantorausdrucks $x \equiv \bigcirc_p\, y$ mindestens so viele Aktionen akzeptiert wie der abstrakte Zweig y. Mit Hilfe des Synchronisationsoperators lassen sich jedoch Beispiele konstruieren, für die dies nicht zutrifft.[35]

Zum anderen war es möglich, die Übersichtlichkeit und Verständlichkeit des Programms gegenüber den früheren prototypischen Implementierungen zu verbessern (die zum Teil noch direkt in C entwickelt wurden), was nicht zuletzt auf die Verwendung der Programmiersprache CH zurückzuführen ist, mit deren Hilfe sich die mathematischen Definitionen und Konzepte (wie z. B. „variante Tupel“, Mengen und Multimengen sowie partielle Funktionen) fast eins zu eins implementierungstechnisch umsetzen ließen.

[35] Im Beispiel $y \equiv a(\omega_1) \circ (b(p) - a(p))$ akzeptiert der Ausdruck $y_p^{\omega_1} \equiv a(\omega_1) \circ \big(b(\omega_1) - a(\omega_1)\big)$ die Aktion $a(\omega_1)$ erst nach Ausführung von $b(\omega_1)$, während sie vom Ausdruck y bereits im ersten Schritt akzeptiert wird, weil in diesem Fall der rechte Zweig der Synchronisation keine Aussage über $a(\omega_1)$ macht.

Kapitel 5

Praktischer Einsatz von Interaktionsausdrücken

5.1 Einleitung

Nachdem Interaktionsausdrücke und -graphen in den vorangegangenen Kapiteln ausführlich beschrieben, theoretisch untersucht und auch praktisch implementiert worden sind, soll im vorliegenden Kapitel genauer auf ihre konkreten Einsatzmöglichkeiten eingegangen werden.

Da für Interaktionsausdrücke sowohl eine präzise formale Semantik als auch eine effiziente Implementierung existiert, können sie einerseits als Beschreibungstechnik zur formalen *Spezifikation* oder *Dokumentation* von Systemeigenschaften und andererseits als Hilfsmittel zur realen *Implementierung* nebenläufiger Systeme verwendet werden. In dieser Hinsicht unterscheiden sie sich von vielen verwandten Ansätzen, bei denen oft nur *einer* dieser Aspekte (in der Regel der erste) im Vordergrund steht, während der andere entweder gar nicht oder nur ansatzweise unterstützt wird.

Beispielsweise wurden Formalismen wie Ereignis- und Flußausdrücke (vgl. § 6.2.3), Life cycles [Coleman94], Sequence diagrams [Zave85], Statecharts [Harel87] oder auch Prozeßalgebren (vgl. § 6.3) primär als *Spezifikationssprachen* entworfen, die typischerweise in den sogenannten „frühen Phasen“ des Software-Engineering eingesetzt werden, aber meist keine direkte Ausführung der erstellten Spezifikationen erlauben. Da Interaktionsausdrücke eine vergleichbare oder sogar höhere Ausdrucksmächtigkeit als die meisten dieser Formalismen besitzen (vgl. Kapitel 6, insbesondere § 6.2.5.2), können sie für derartige Anwendungsbereiche ebenfalls eingesetzt werden, wobei die Verfügbarkeit einer graphischen Notation einschließlich eines zugehörigen Editors (vgl. Anhang C) besonders vorteilhaft ist.

Neben der reinen *Spezifikation* von Synchronisationsbedingungen wird beim Einsatz von Interaktionsausdrücken jedoch auch die *implementierungstechnische Umsetzung* der erstellten Spezifikationen unterstützt, sofern die in Kapitel 4 beschriebene Implementierung geeignet in eine „Prozeßausführungsumgebung“ integriert wird. Hierfür wird diese „Kernimplementierung“ zunächst (§ 5.2) zu einem vollständigen *Interaktionsmanager* ausgebaut, dessen Dienste über geeignete Schnittstellen und Protokolle (§ 5.2.1 bis § 5.2.6) genutzt werden können. Außerdem wird erläutert, wie das grundsätzliche Problem der *Skalierbarkeit* durch den Einsatz mehrerer Interaktionsmanager (§ 5.2.7) und ggf. eine *Partitionierung* von Interaktionsgraphen (§ 5.2.8) bewältigt werden kann. Die Beschreibung einer konkreten Interaktionsmanager-Implementierung (§ 5.2.9) sowie Überlegungen zum Wiederanlauf nach Systemausfällen (engl. recovery) (§ 5.2.10) runden diesen grundlegenden Abschnitt ab.

Anschließend wird anhand von zwei wichtigen Einsatzgebieten von Interaktionsausdrücken − Synchronisation paralleler Programme (§ 5.3) und Definition und Implementierung von Workflow-Geflechten (§ 5.4 und § 5.5) − erläutert, wie nebenläufige Systeme mit Hilfe von Interaktionsgraphen sinnvoll entwickelt werden können und wie der zuvor beschriebene Interaktionsmanager konkret zur Implementierung der spezifizierten Synchronisationsbedingungen eingesetzt werden kann. Zur Implementierung von Workflow-Geflechten werden hierfür zwei alternative Ansätze − Adaption von Arbeitslistenprogrammen (§ 5.5.2) und Adaption von Workflow-Ausführungseinheiten (§ 5.5.3) − vorgestellt und miteinander verglichen (§ 5.5.4).

Anmerkung: Obwohl die Synchronisation paralleler Programme nicht zum eigentlichen Thema dieser Arbeit gehört, stellt sie doch ein interessantes und wichtiges Einsatzgebiet von Interaktionsausdrücken dar und wird deshalb kurz vorgestellt. Außerdem können auf diese Weise einige interessante Parallelen zwischen nebenläufigen Programmen und Workflow-Geflechten (d. h. Systemen nebenläufiger Workflows) aufgezeigt werden, die dazu führen, daß beide auf ähnliche Art und Weise entwickelt werden können (vgl. § 5.3.3 und § 5.4.1).

5.2 Interaktionsmanager

5.2.1 Konzeption

Um sicherzustellen, daß die durch einen Interaktionsgraphen oder -ausdruck[1] x spezifizierte Integritätsbedingung in der realen Welt tatsächlich eingehalten wird, muß die Ausführung von Aktionen mit einer übergeordneten Instanz, einem sogenannten *Interaktionsmanager*, abgestimmt werden. Dieser muß die Ausführung von Aktionen einerseits *verfolgen*, d. h. den Graphen anhand der ausgeführten Aktionen *zielgerichtet durchlaufen* (vgl. § 2.4.2) bzw. ausgehend vom initialen Zustand des Ausdrucks x bei jeder Aktionsausführung einen entsprechenden *Zustandsübergang* durchführen (vgl. § 4.5.1.4). Andererseits muß der Interaktionsmanager die Ausführung von Aktionen *kontrollieren*, d. h. Aktionsausführungen verhindern, die im aktuellen Zustand des Ausdrucks nicht zulässig sind. Um diese beiden Aufgaben erfüllen zu können, muß der Interaktionsmanager von den aktionsausführenden Instanzen (den sogenannten *Klienten*) einerseits über jede Aktionsausführung *informiert* werden und andererseits vor jeder Aktionsausführung *um Erlaubnis gefragt* werden.

Für diese Kommunikation zwischen Interaktionsmanager und Klienten, die in Abb. 5.1 schematisch dargestellt ist, dienen die im folgenden beschriebenen *Koordinationsprotokolle*.[2] Sie spezifizieren jeweils eine Menge von *Nachrichten*, bei denen es sich um *Anfragen* eines Klienten, vorläufige oder endgültige *Antworten* des Interaktionsmanagers oder *Bestätigungen* des Klienten handeln kann. Jede Nachricht besteht aus einem *Schlüsselwort* und einer zugehörigen Aktion a, auf die sich die Anfrage, Antwort oder Bestätigung bezieht. Der Zeitraum zwischen dem Empfangen oder Versenden einer Nachricht und dem Versenden oder Empfangen der darauffolgenden Nachricht wird als *Phase* eines Koordinationsprotokolls bezeichnet. Dementsprechend besteht ein Protokoll, bei dem pro geplanter Aktionsausführung insgesamt n Nachrichten zwischen Klient und Interaktionsmanager ausgetauscht werden, aus $n-1$ Phasen.

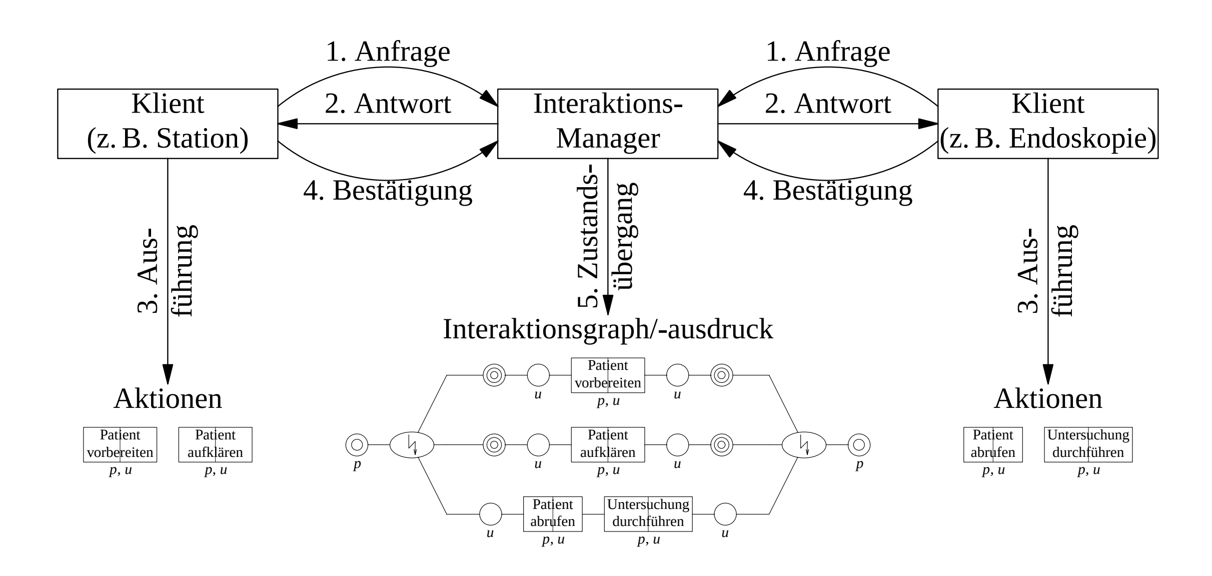


Abbildung 5.1: Kommunikation zwischen Interaktionsmanager und Klienten

Bei den aktionsausführenden Klienten kann es sich z. B. um Prozesse eines verteilten Programms (vgl. § 5.3.2), die zentrale Ausführungseinheit eines oder mehrerer Workflow-Management-Systeme (§ 5.5.3) oder auch um Arbeitslistenprogramme einzelner Benutzer handeln (§ 5.5.2). Damit ein ein-

[1] Die Begriffe (Interaktions-)Graph und (Interaktions-)Ausdruck werden im folgenden synonym verwendet, da jeder Graph eineindeutig einem Ausdruck entspricht.
[2] Konkret bezieht sich die Abbildung auf das *zweiphasige Koordinationsprotokoll* (vgl. § 5.2.3).

zelner Interaktionsmanager durch eine große Zahl von Klienten nicht überlastet wird, ist es u. U. sinnvoll, *mehrere* Manager einzusetzen, die zwar möglichst *unabhängig* voneinander agieren sollten, im Zweifelsfall aber doch wechselseitig koordiniert werden müssen. Hierfür werden ebenfalls geeignete Koordinationsprotokolle eingesetzt (vgl. § 5.2.7.3).

5.2.2 Einphasiges Koordinationsprotokoll

5.2.2.1 Beschreibung

Die einfachste Form der Kommunikation zwischen einem Klienten und dem Interaktionsmanager besteht aus einer *Anfrage* des Klienten, ob eine bestimmte Aktion momentan ausgeführt werden darf oder nicht, und einer zugehörigen *Antwort* des Managers. Da ein Klient häufig nur an einer *positiven* Antwort interessiert ist, kann er eine Anfrage auch in Form eines „Antrags“ stellen, der vom Manager erst dann „bewilligt“ wird, wenn die angefragte Aktion ausgeführt werden darf.

Tabelle 5.2 zeigt die Schlüsselwörter (oder *Nachrichtentypen*) des entsprechenden *einphasigen Koordinationsprotokolls*, bei dem pro geplanter Aktionsausführung zwei Nachrichten zwischen Klient und Interaktionsmanager ausgetauscht werden.

Anfragen	Antworten
Ask	Accept
Wait	Reject

Tabelle 5.2: Nachrichtentypen beim einphasigen Koordinationsprotokoll

Die Nachrichten Ask und Wait fragen beide beim Interaktionsmanager an, ob eine bestimmte Aktion a im aktuellen Zustand des Ausdrucks x zulässig ist. Ist dies der Fall, antwortet der Manager mit einer entsprechenden Accept-Nachricht, die besagt, daß die Aktion a ausgeführt werden *darf*; außerdem geht der Manager davon aus, daß a anschließend tatsächlich ausgeführt *wird*, d. h. er führt intern bereits einen entsprechenden *Zustandsübergang* durch.

Ist die Aktion momentan nicht zulässig, wird eine Ask-Anfrage mit einer entsprechenden Reject-Nachricht beantwortet, während eine Wait-Anfrage (die dem oben erwähnten „Antrag“ entspricht) zunächst unbeantwortet bleibt. Sie wird vom Interaktionsmanager solange aufbewahrt, bis sie schließlich mit einer Accept-Nachricht positiv beantwortet werden kann.

Für einen Klienten hat eine Anfrage mittels Ask zwei potentielle Vorteile: Zum einen wird sie sofort beantwortet, zum anderen ist der Klient im Falle einer negativen Antwort (Reject) zu nichts verpflichtet. Die Antwort auf eine Wait-Anfrage kann hingegen sehr lange ausbleiben, und der Klient ist verpflichtet, nach ihrem Eintreffen die Aktion a tatsächlich auszuführen. Sofern diese Verpflichtung für einen Klienten kein Problem darstellt, hat Wait gegenüber Ask jedoch den Vorteil, daß der Klient für jede Aktion, die er ausführen möchte, nur *einmal* eine Wait-Anfrage stellen muß, während er bei Verwendung von Ask seine Anfrage u. U. mehrmals wiederholen muß.

Abbildung 5.3 zeigt zwei Interaktionsgraphen[3], die die Abfolge der verschiedenen Nachrichtentypen (für eine bestimmte Aktion a und einen einzelnen Klienten) sowie ihr Zusammenspiel mit dem Zustandsübergang auf Managerseite (unterer Graph) und der tatsächlichen Aktionsausführung auf Klien-

[3] Analog zu einer Metasprache, d. h. einer Sprache, die zur *Beschreibung* einer anderen Sprache verwendet wird und ansonsten nichts mit dieser Sprache zu tun hat, könnte man die Graphen als *Metagraphen* bezeichnen, da sie nichts mit dem vom Interaktionsmanager zu verarbeitenden Graphen x zu tun haben, sondern lediglich das Zusammenspiel von Klient und Interaktionsmanager beschreiben sollen. Sie stellen somit eine der in § 5.1 erwähnten Anwendungen von Interaktionsgraphen als Beschreibungstechnik zur formalen Spezifikation oder Dokumentation von Systemeigenschaften dar.

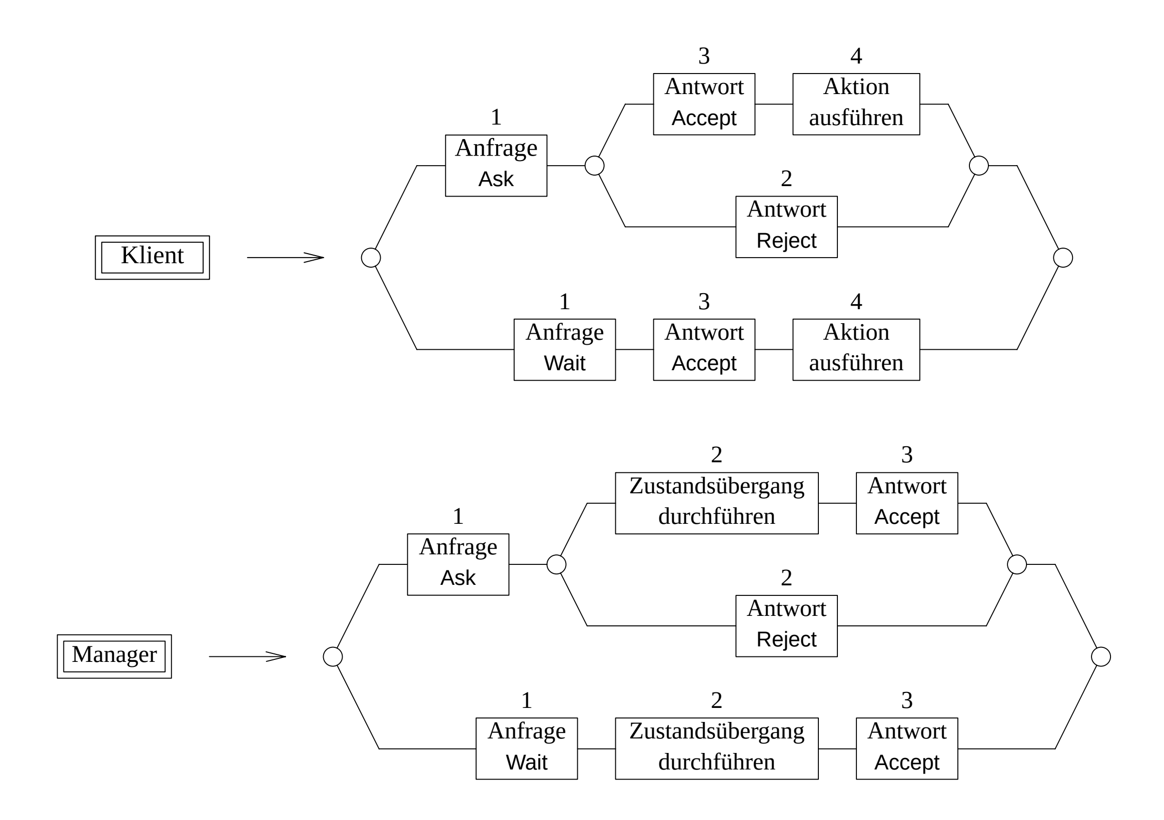


Abbildung 5.3: Ablauf des einphasigen Koordinationsprotokolls

tenseite (oberer Graph) noch einmal verdeutlichen:[4] Der Klient kann die Erlaubnis zur Ausführung der Aktion a entweder mittels Ask oder mittels Wait einholen. Im ersten Fall erhält er *sofort* eine entsprechende Accept- oder Reject-Nachricht als Antwort, während er im zweiten Fall *irgendwann* eine Accept-Nachricht erhält.[5] *Vor* dem Versenden einer Accept-Nachricht führt der Interaktionsmanager einen Zustandsübergang für die Aktion a durch, während der Klient *nach* Erhalt dieser Nachricht die Aktion tatsächlich ausführt.

5.2.2.2 Kritik

Diese zeitliche Entkopplung von Zustandsübergang einerseits und tatsächlicher Aktionsausführung andererseits stellt einen wesentlichen Schwachpunkt des einphasigen Koordinationsprotokolls dar, der es für viele Anwendungen unbrauchbar macht. Wie das folgende Beispiel zeigt, ist es dadurch tatsächlich möglich, daß Aktionen in der realen Welt in einer *unzulässigen Reihenfolge* ausgeführt werden.

Gegeben sei der Ausdruck $x \equiv a - b$, der im ersten Schritt die Aktion a und im zweiten Schritt die Aktion b erlaubt. Werden beide Aktionen von unterschiedlichen Klienten angefragt, so verschickt der Interaktionsmanager zunächst eine Accept-Antwort für a, nachdem er den entsprechenden Zustandsübergang durchgeführt hat. Da die Aktion b im resultierenden Zustand zulässig ist, verschickt er unmittelbar anschließend auch eine Accept-Antwort für b (ebenfalls nach einem entsprechenden Zustandsübergang). Wenn nun der Klient, der b angefragt hat, seine Accept-Antwort früher erhält oder schneller verarbeitet als der Klient, der a angefragt hat, so kann es passieren, daß die Aktion b in der realen Welt *vor* der Aktion a ausgeführt wird, obwohl dies durch den Ausdruck x gerade verhindert werden soll.

[4] Man stelle sich vor, daß die beiden Graphen durch eine Kopplung verknüpft sind, so daß gemeinsame (Meta-)Aktionen gleichzeitig durchlaufen werden müssen. Da es sich bei diesen gerade um die ausgetauschten Nachrichten handelt, stellen diese die Synchronisationspunkte zwischen Klient und Manager dar. Die Zahlen verdeutlichen die zeitliche Abfolge der einzelnen Schritte.

[5] „Irgendwann“ könnte u. U. auch „nie“ sein, wenn der Ausdruck x keinen Zustand erreicht, in dem die Aktion a zulässig ist.

Anmerkung: In Anwendungen wie z. B. der Synchronisation paralleler Programme, in denen der tatsächliche Ausführungszeitpunkt einer Aktion − d. h. in diesem Zusammenhang das Starten oder Beenden einer Prozedur (vgl. § 5.3.2.1) − nicht unmittelbar meß- oder beobachtbar ist, kann man aber u. U. vereinbaren, daß eine Aktion formal genau dann ausgeführt wird, wenn der Interaktionsmanager den entsprechenden Zustandsübergang durchführt. Daß beispielsweise die erste Instruktion einer angefragten Prozedur in Wirklichkeit etwas später ausgeführt wird, spielt hier keine Rolle, da eine solche Verzögerung genausogut durch eine vorübergehende Suspendierung des Prozesses durch das Betriebssystem bedingt sein könnte. Für derartige Anwendungen ist das einphasige Koordinationsprotokoll also vollkommen ausreichend.

5.2.3 Zweiphasiges Koordinationsprotokoll

5.2.3.1 Beschreibung

Will man den zuvor erläuterten Mangel des einphasigen Koordinationsprotokolls beheben, um es auch für Anwendungen brauchbar zu machen, in denen die Ausführungszeitpunkte von Aktionen exakt beobachtet werden können, so muß man die Durchführung des Zustandsübergangs und die tatsächliche Ausführung der Aktion in der realen Welt miteinander synchronisieren. Zu diesem Zweck wird das einphasige Protokoll im folgenden zu einem *zweiphasigen Koordinationsprotokoll* erweitert, bei dem pro geplanter Aktionsausführung bis zu drei Nachrichten zwischen Klient und Interaktionsmanager ausgetauscht werden.

Wie in Tab. 5.4 dargestellt, gibt es bei einem zweiphasigen Protokoll neben Anfragen von Klienten und Antworten des Managers noch zusätzliche *Bestätigungen* seitens der Klienten. Die Anfragen Ask und Wait sowie die Antwort Reject besitzen hierbei dieselbe Bedeutung wie beim einphasigen Protokoll. Auch die Antwort Accept signalisiert ebenso wie dort, daß die angefragte Aktion ausgeführt werden *darf*; der zugehörige Zustandsübergang wird vom Manager aber erst durchgeführt, wenn der Klient die *tatsächliche Ausführung* der Aktion in der realen Welt mittels einer Exec-Nachricht bestätigt hat. Damit Zustandsübergang und reale Aktionsausführung hierbei tatsächlich *synchronisiert* erfolgen, bleibt der Interaktionsmanager zwischen dem Versenden der Accept-Antwort und dem Eintreffen der Exec-Bestätigung *blockiert*, d. h. er verarbeitet in dieser Phase keine anderen Klientenanfragen.

Phase 1		Phase 2
Anfragen	Antworten	Bestätigungen
Ask Wait	Accept Reject	Exec Undo

Tabelle 5.4: Nachrichtentypen beim zweiphasigen Koordinationsprotokoll

Natürlich kann auch bei diesem Protokoll nicht verhindert werden, daß Aktionsausführung und Zustandsübergang zeitlich mehr oder weniger auseinanderliegen. Es wird jedoch gewährleistet, daß während dieser „kritischen Phase“ weder eine andere Aktion noch ein anderer Zustandsübergang ausgeführt wird. Somit ist tatsächlich sichergestellt, daß die Aktionen in der realen Welt in *derselben Reihenfolge* wie die entsprechenden Zuständsübergänge ausgeführt werden.

Analog zu Abb. 5.3, zeigen die Graphen in Abb. 5.5 den Ablauf des zweiphasigen Koordinationsprotokolls für den Anfragetyp Ask noch einmal im Zusammenhang (vgl. auch Abb. 5.1; entsprechende Graphen für den Anfragetyp Wait würde man erhalten, wenn man die Zweige mit dem Antworttyp Reject entfernt). Wie man sieht, besteht das Protokoll nur dann wirklich aus zwei Phasen, wenn der Manager die Anfrage des Klienten mit Accept beantwortet, da nur in diesem Fall eine Bestätigung des

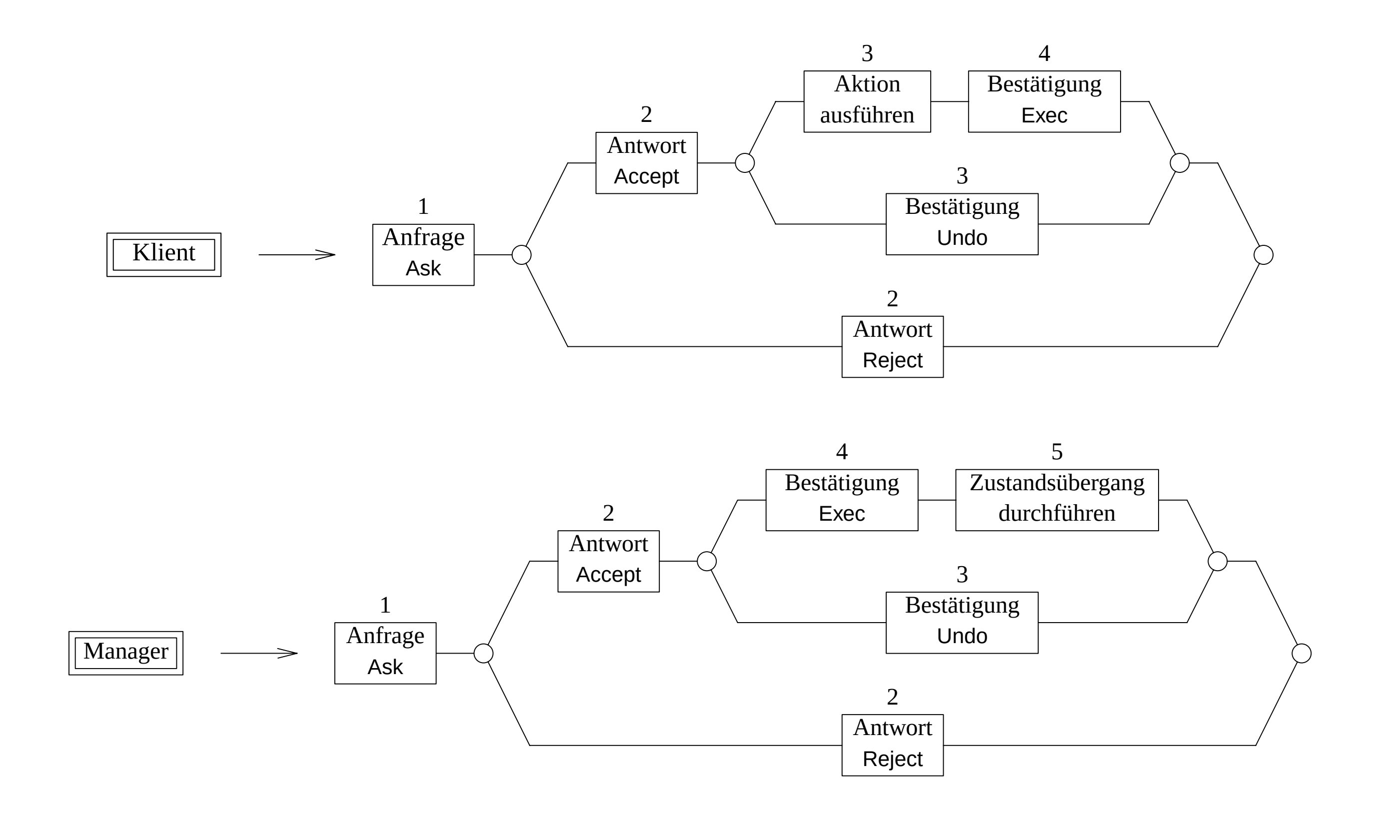


Abbildung 5.5: Ablauf des zweiphasigen Koordinationsprotokolls

Klienten erforderlich ist. Die Graphen zeigen außerdem, daß ein Klient eine Accept-Antwort des Managers auch mit einer Undo-Nachricht quittieren kann, wenn er die angefragte Aktion aus irgendwelchen Gründen doch nicht ausführen möchte. In diesem Fall führt der Manager keinen Zustandsübergang durch. Diese Option wird in § 5.2.7.3 zur Koordination mehrerer Interaktionsmanager verwendet.

5.2.3.2 Kritik

Obwohl das zweiphasige Koordinationsprotokoll, wie erläutert, korrekte Ausführungsreihenfolgen von Aktionen gewährleistet, besitzt es doch − ebenso wie das bekannte Zwei-Phasen-Commit-Protokoll (2PC) verteilter Datenbanksysteme [Dadam96, Özsu91] − einen gravierenden Nachteil: Nach dem Versenden einer Accept-Antwort an einen Klienten ist der Interaktionsmanager so lange blockiert, bis er eine entsprechende Exec- oder Undo-Bestätigung des Klienten erhält. Sollte diese Nachricht aus irgendwelchen Gründen nicht oder nicht in angemessener Zeit eintreffen, ist der Manager vollkommen *handlungsunfähig*, da er ohne Kenntnis dieser Nachricht weder den Zustandsübergang für die Aktion *a* durchführen noch eigenmächtig entscheiden kann, ihn nicht durchzuführen. Durch das Versenden der Accept-Nachricht hat er quasi alle Rechte, eine derartige Entscheidung zu treffen, an den Klienten abgetreten und muß nun dessen Entscheidung abwarten, wie lange dies auch dauern mag.[6] Andererseits kann der Manager keine anderen Anfragen bearbeiten, solange diese Entscheidung nicht gefällt ist.

Auch wenn man davon ausgehen kann, daß ein Klient Entscheidungen dieser Art nicht mutwillig verzögert, so kann doch aufgrund von Programm- oder Rechnerabstürzen einerseits und Netzwerkstörungen oder -unterbrechungen andererseits jederzeit ein derartiger Fall eintreten. Da der Interaktionsmanager aber vor jeder Aktionsausführung um Erlaubnis gefragt werden muß, blockiert ein hand-

[6] Vergleicht man das zweiphasige Koordinationsprotokoll mit einem Zwei-Phasen-Commit-Protokoll, so erkennt man, daß der Klient quasi die Rolle des Commit-Koordinators spielt, während der Interaktionsmanager einem gewöhnlichen Teilnehmer entspricht.

lungsunfähiger Interaktionsmanager zwangsläufig auch jede weitere Aktionsausführung in einem System. Bedenkt man außerdem, daß es sich bei einem Klienten u. U. um ein lokales Arbeitslistenprogramm eines Benutzers handelt (vgl. § 5.5.2), dessen Bürocomputer (PC) jederzeit absichtlich oder versehentlich ausgeschaltet oder vom Netzwerk getrennt werden kann, so wird klar, daß die Handlungsfähigkeit des Interaktionsmanagers − der in einem Gesamtsystem eine äußerst wichtige und Performance-kritische Komponente darstellt − nicht von der Zuverlässigkeit und Verfügbarkeit eines vergleichsweise unwichtigen Programms abhängen darf.

Aus diesen Gründen wird das zuvor beschriebene zweiphasige Koordinationsprotokoll im folgenden um eine dritte Phase erweitert, die es dem Interaktionsmanager erlaubt, den „schwarzen Peter“ rechtzeitig an den Klienten abzugeben, um im Zweifelsfall unabhängig von diesem handlungsfähig zu bleiben.

5.2.4 Dreiphasiges Koordinationsprotokoll

5.2.4.1 Beschreibung

Tabelle 5.6 und Abb. 5.7 zeigen die Nachrichtentypen und ihr Zusammenspiel beim *dreiphasigen Koordinationsprotokoll*, bei dem pro geplanter Aktionsausführung bis zu vier Nachrichten zwischen Klient und Interaktionsmanager ausgetauscht werden. Im Gegensatz zum ein- und zweiphasigen Protokoll, stellt der Nachrichtentyp Accept hier nur eine *vorläufige* Antwort des Interaktionsmanagers an den Klienten dar, die er später entweder mit einer Commit-Antwort bestätigen oder aber mit einer Abort-Antwort widerrufen kann. Für den Klienten bedeutet dies, daß er nach Erhalt einer Accept-Nachricht die angefragte Aktion zwar zunächst ausführen darf (und muß), sie später aber unter Umständen wieder *rückgängig machen* muß. Aus diesem Grund ist es für den Klienten ratsam, beim Eintreffen der Accept-Nachricht eine *Transaktion* zu beginnen, die er später entweder erfolgreich abschließen oder aber *zurücksetzen* kann. Sobald er (innerhalb dieser Transaktion) die Aktion ausgeführt und die Transaktion *gesichert* hat (siehe unten), sendet er dem Manager eine Exec-Bestätigung und wartet dann auf dessen endgültige Antwort.

Phase 1		Phase 2	Phase 3
Anfragen	(vorläufige) Antworten	Bestätigungen	endgültige Antworten
Ask Wait	Accept Reject	Exec Undo	Commit Abort

Tabelle 5.6: Nachrichtentypen beim dreiphasigen Koordinationsprotokoll

Auf diese Weise wird „der Spieß umgedreht“, weil jetzt der Klient so lange handlungsunfähig ist, bis er vom Manager eine Commit- oder Abort-Nachricht erhält. Sollte diese ausbleiben oder sich verzögern, kann der Klient nur warten und ggf. periodisch beim Manager nachfragen. In aller Regel ist eine solche Blockierung eines Klienten jedoch weit weniger problematisch als die des Managers.

Sollte die Exec-Bestätigung des Klienten ausbleiben oder sich zu lange verzögern, kann der Interaktionsmanager seinen Blockierungszustand jederzeit *eigenmächtig* beenden, indem er dem Klienten eine Abort-Nachricht schickt.

5.2.4.2 Anmerkungen

Ähnlich wie das zweiphasige Koordinationsprotokoll, kann auch das dreiphasige Protokoll durch eine Reject-Antwort des Managers oder eine Undo-Bestätigung des Klienten „abgekürzt“ werden. In diesen Fällen besteht das Protokoll lediglich aus ein bzw. zwei Phasen.

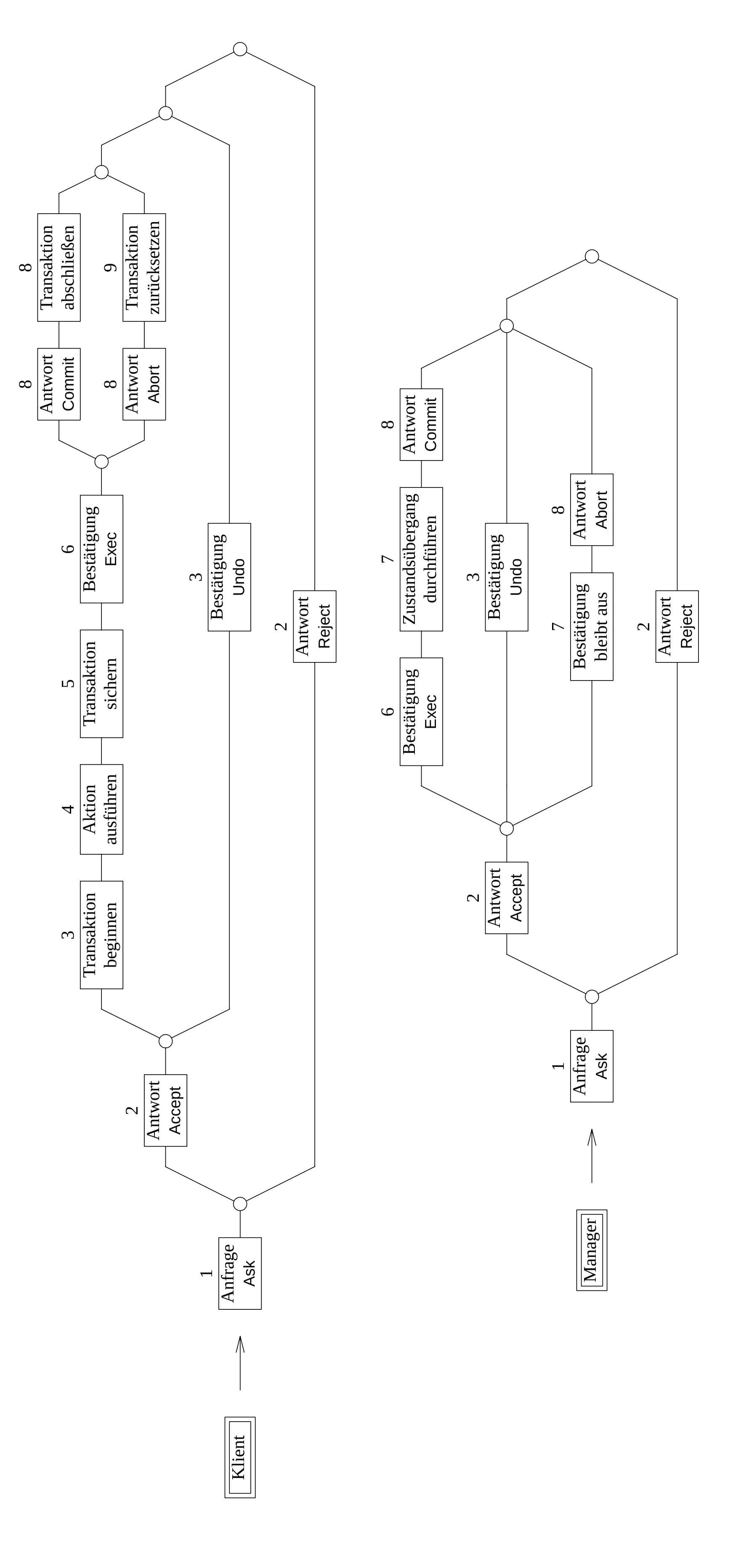


Abbildung 5.7: Ablauf des dreiphasigen Koordinationsprotokolls

Wird das Protokoll vollständig durchlaufen, so entsprechen die Phasen 2 und 3 den beiden Phasen eines Zwei-Phasen-Commit-Protokolls, bei dem der Interaktionsmanager die Rolle des *Commit-Koordinators* spielt: Die Nachricht Accept, die den Klienten auffordert, die Aktion innerhalb einer Transaktion auszuführen und diese zu sichern, ist mit der *Prepare-to-commit*-Aufforderung des 2PC-Protokolls vergleichbar, während die Exec-Nachricht des Klienten einer *Ready-to-commit*-Bestätigung entspricht. In diesem Zustand muß der Klient einerseits in der Lage sein, die Transaktion − auch im Falle eines Systemausfalls o. ä. − erfolgreich abzuschließen, wenn der Interaktionsmanager dies durch Senden einer Commit-Antwort verlangt; andererseits muß der Klient die Transaktion noch zurücksetzen können, wenn er vom Manager eine Abort-Nachricht erhält.

Handelt es sich bei den auszuführenden Aktionen um das Starten oder Beenden von Workflowschritten, so werden diese von einem Workflow-Management-System normalerweise als echte Datenbank-Transaktionen ausgeführt, die man zumeist über eine geeignete Schnittstelle (z. B. X/OPEN XA [Gray93, Dadam96]) in den benötigten Ready-to-commit-Zustand versetzen kann. Für andere Arten von Anwendungen, deren Aktionsausführungen nichts mit Datenbank-Transaktionen zu tun haben, muß jeweils überprüft werden, ob und ggf. wie ein derartiger „Schwebezustand“ des Klienten erreicht werden kann, in dem eine Aktion einerseits bereits ausgeführt ist, andererseits aber auch noch rückgängig gemacht werden kann. Sollte dies prinzipiell nicht möglich sein, so muß man zwangsläufig auf das ein- oder zweiphasige Koordinationsprotokoll − mit den bereits erläuterten Nachteilen − ausweichen.

5.2.5 Abonnierte Nachrichten

5.2.5.1 Motivation

Die bisher betrachteten Koordinationsprotokolle können verwendet werden, wenn ein Klient beabsichtigt, eine bestimmte Aktion *tatsächlich* auszuführen. Sie sind jedoch weniger gut geeignet, wenn er sich nur unverbindlich erkundigen möchte, ob eine Aktion momentan ausgeführt werden *dürfte*. Zwar könnte ein Klient das zwei- oder dreiphasige Koordinationsprotokoll für diesen Zweck „mißbrauchen“, indem er eine Ask-Anfrage an den Interaktionsmanager stellt und im Fall einer Accept-Antwort eine Undo-Bestätigung zurückschickt. Allerdings würde er auf diese Weise den Manager unnötig belasten, da dieser zwischen dem Versenden seiner Accept-Antwort und dem Eintreffen der Undo-Bestätigung normalerweise blockiert ist.

Außerdem kann ein Klient mit dieser Methode immer nur den *augenblicklichen* Status einer Aktion in Erfahrung bringen, der im nächsten Augenblick schon wieder veraltet sein könnte. Handelt es sich bei einem Klienten beispielsweise um ein Arbeitslistenprogramm eines Workflow-Management-Systems, so wäre es wünschenswert, wenn er seinem Benutzer jederzeit den aktuellen Status aller für ihn interessanten Aktionen anzeigen könnte. Hätte der Klient hierfür nur die bislang betrachteten Nachrichtentypen zur Verfügung, so müßte er die zuvor beschriebene Prozedur für jede Aktion relativ häufig wiederholen, um jederzeit möglichst aktuelle Statusinformation zu besitzen. Dies hätte zur Folge, daß die Anzahl dieser unverbindlichen Anfragen, die den Interaktionsmanager jedesmal unnötig blockieren, die Zahl der „wirklichen“ Anfragen (bei denen tatsächlich eine Aktion ausgeführt werden soll) vermutlich um ein Vielfaches übersteigen würde.

5.2.5.2 Beschreibung

Um das daraus resultierende Kommunikationsaufkommen und die unnötige Belastung des Managers zu vermeiden, sollte der Interaktionsmanager seinen Klienten anbieten, Mitteilungen über Status-*Änderungen* bestimmter Aktionen zu *abonnieren*. Tabelle 5.8 und Abb. 5.9 zeigen die hierfür benötigten Nachrichtentypen und ihr Zusammenspiel:[7] Wenn ein Klient daran interessiert ist, stets über den aktuellen Status einer Aktion a informiert zu werden, so bringt er dies durch eine entsprechende

[7] Da der Graph (im Gegensatz zu früheren Graphen) nur die zwischen Klient und Manager ausgetauschten Nachrichten darstellt, ist er für Klient und Manager identisch.

Bestellung	Mitteilungen	Kündigung
Subscribe	Permit Forbid	Cancel

Tabelle 5.8: Nachrichtentypen beim Abonnementprotokoll

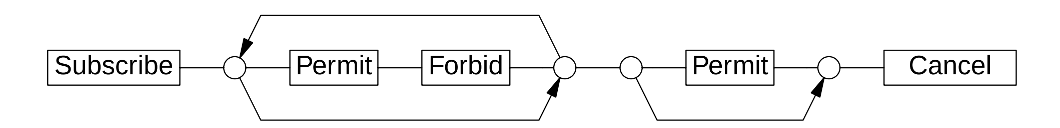


Abbildung 5.9: Ablauf des Abonnementprotokolls

Subscribe-Nachricht an den Interaktionsmanager zum Ausdruck. In der Folge erhält er vom Manager unaufgefordert immer dann eine Permit- bzw. Forbid-Nachricht, wenn sich der Status der Aktion von unzulässig auf zulässig bzw. umgekehrt geändert hat. Ist die Aktion zum Zeitpunkt der Subscribe-Nachricht gerade zulässig, wird darüber hinaus sofort eine erste Permit-Nachricht verschickt; ist die Aktion gerade nicht zulässig, bekommt der Klient so lange keine Mitteilung, bis die Aktion zum ersten Mal zulässig ist. Ist der Klient am Status einer Aktion a nicht mehr interessiert, kann er das Abonnement jederzeit durch eine Cancel-Nachricht kündigen, unabhängig davon, ob er bereits eine oder mehrere Permit- und/oder Forbid-Nachrichten erhalten hat.

Um Mißverständnissen vorzubeugen, sollte an dieser Stelle betont werden, daß eine Permit-Nachricht des Managers rein informativen Charakter besitzt − die entsprechende Aktion *dürfte* momentan ausgeführt werden − und vom Klienten nicht als tatsächliche Ausführungserlaubnis für eine Aktion a interpretiert werden darf. Um eine Aktion tatsächlich auszuführen, muß nach wie vor eines der zuvor beschriebenen Koordinationsprotokolle durchlaufen werden. Allerdings kann der Klient eine Permit-Mitteilung natürlich zum Anlaß nehmen, *jetzt* eine Ask-Anfrage bzgl. a zu stellen, weil diese dann mit großer Wahrscheinlichkeit erfolgreich sein wird, d. h. vom Manager mit Accept beantwortet werden wird.

5.2.6 Sonderbehandlung unbekannter Aktionen

Bisher wurde ein Interaktionsgraph bzw. -ausdruck als *abgeschlossenes* System aufgefaßt: Eine Aktion a darf ausgeführt werden, wenn sie im Graphen momentan durchlaufen werden kann bzw. wenn der aktuelle Zustand s des Ausdrucks x durch den Zustandsübergang $\tau_a(s)$ in einen gültigen Folgezustand übergeht; andernfalls darf a nicht ausgeführt werden. Insbesondere dürfen *unbekannte* Aktionen, d. h. Aktionen, die im Graphen bzw. Ausdruck nicht vorkommen, *niemals* ausgeführt werden, weil sie nie durchlaufen werden können bzw. weil ihr Zustandsübergang stets einen ungültigen Folgezustand liefert (vgl. auch § 3.4.4.2).

Diese Sichtweise entspricht dem in wissensbasierten Systemen als *closed world assumption* oder auch *negation as failure* bezeichneten Prinzip, nach dem eine Aussage als falsch angesehen wird, wenn ihre Richtigkeit aus der vorhandenen Wissensbasis nicht abgeleitet werden kann [Reiter78, Clark78]. Ein Prolog-Interpreter arbeitet beispielsweise nach diesem Prinzip, weil er eine Anfrage mit *no* beantwortet, wenn er sie aus den vorhandenen Fakten und Regeln nicht ableiten kann [Clocksin94]. Bei einer solchen Sichtweise wird implizit davon ausgegangen, daß sowohl die Wissensbasis als auch die angewandten Inferenzmechanismen *vollständig* sind.

Würde man nun beispielsweise einem Prolog-Programm, das biographische Daten berühmter Maler verwaltet, die Frage stellen, ob J. S. Bach im Jahre 1685 geboren wurde, so würde man fälschlicherweise die Antwort *no* erhalten, weil die Faktenbasis des Programms keine Informationen über Komponisten enthält. In ähnlicher Weise würde ein Klient von einem Interaktionsmanager die Antwort

Reject erhalten, wenn er eine Aktion anfragt, die der Manager nicht kennt, weil sie in seinem Ausdruck x nicht vorkommt. Ebenso wie es nun unsinnig wäre, die Aussage „J. S. Bach ist 1685 geboren“ als falsch zu verwerfen, nur weil sie aus einer bestimmten Wissensbasis nicht abgeleitet werden kann, wäre es auch unangebracht, die Ausführung einer Aktion a prinzipiell zu verbieten, nur weil sie in einem bestimmten Interaktionsgraphen nicht enthalten ist.

Wollte man diese abgeschlossene Sicht eines Graphen tatsächlich vertreten, so wäre ein Graphautor gezwungen, sämtliche in der realen Welt ausführbaren Aktionen bei der Formulierung eines Graphen zu berücksichtigen, was zweifellos unrealistisch wäre. Vielmehr sollte ein Interaktionsmanager bei der Interpretation seines Ausdrucks x eine *open world assumption* zugrundelegen, die damit rechnet, daß es neben den im Ausdruck x vorkommenden Aktionen auch noch andere Aktionen gibt, über die x *keine Aussage macht*. Konkret bedeutet dies, daß ein Interaktionsmanager eine Anfrage bzgl. einer Aktion a nicht nur mit Accept oder Reject, sondern auch mit dem Nachrichtentyp Unknown beantworten kann (vgl. Tab. 5.10), wenn ihm die Aktion unbekannt ist und er daher keine Aussage über sie machen kann. Ähnlich wie beim Antworttyp Reject, wird in diesem Fall kein Zustandsübergang durchgeführt, und der Manager erwartet keine Bestätigung seitens des Klienten.

Phase 1		Phase 2	Phase 3
Anfragen	(vorläufige) Antworten	Bestätigungen	endgültige Antworten
Ask Wait	Accept Reject Unknown	Exec Undo	Commit Abort

Tabelle 5.10: Zusätzlicher Nachrichtentyp Unknown

Für einen Klienten bedeutet eine Antwort vom Typ Unknown jedoch, daß er die fragliche Aktion *jederzeit* − ohne den Manager fragen oder informieren zu müssen − ausführen darf. Um unnötige Anfragen an den Interaktionsmanager zu vermeiden, kann es daher sinnvoll sein, wenn ein Klient a priori weiß, welche Aktionen der Interaktionsmanager kennt und welche nicht, weil er letztere dann eigenmächtig − ohne jegliche Kommunikation mit dem Manager − ausführen kann. Dies ist insbesondere beim Einsatz mehrerer Interaktionsmanager sinnvoll, um den Kommunikationsaufwand sowie die Belastung der einzelnen Manager zu reduzieren (vgl. § 5.2.7.1).

5.2.7 Einsatz mehrerer Interaktionsmanager

5.2.7.1 Motivation

Hat man für eine bestimmte Anwendung mehrere Integritätsbedingungen in Form von Interaktionsgraphen entwickelt, die alle gleichzeitig eingehalten werden sollen, so müssen die einzelnen Graphen gemäß § 2.3.2.2 bzw. § 2.7.5 mit Hilfe einer Kopplung verknüpft werden. Auf diese Weise wird sichergestellt, daß eine Aktion genau dann ausgeführt werden darf, wenn sie in allen Graphen, in denen sie auftritt, gleichzeitig durchlaufen werden kann. Aufgrund der soeben erläuterten *open world assumption* ist außerdem gewährleistet, daß Aktionen, die in keinem Graphen vorkommen, jederzeit ausgeführt werden dürfen.

Würde man nun nicht die Kopplung bzw. Synchronisation $x \equiv x_1 \circ \ldots \circ x_n$ der einzelnen Ausdrücke x_i an einen einzigen Interaktionsmanager übergeben, sondern jeden Ausdruck x_i von einem *eigenen* Manager verwalten lassen, so wäre die Semantik exakt dieselbe: Eine Aktion a darf genau dann ausgeführt werden, wenn sie von allen Managern, *die die Aktion kennen*, gleichzeitig erlaubt wird, d. h. wenn sie in allen Graphen x_i, *in denen sie auftritt*, gleichzeitig durchlaufen werden kann. Manager, die die Aktion a nicht kennen, müssen entweder gar nicht gefragt werden, oder sie würden die

Anfrage mit einer Unknown-Nachricht beantworten, was prinzipiell äquivalent ist. Der Spezialfall einer „global unbekannten“ Aktion, die in keinem der Ausdrücke x_i vorkommt, wird ebenfalls gleich behandelt, da für eine solche Aktion keiner der Manager um Erlaubnis gefragt werden muß.

In praktischen Anwendungen kann die Verwendung mehrerer Interaktionsmanager die Performance und Skalierbarkeit eines Systems u. U. entscheidend verbessern. Hätte man beispielsweise *paarweise disjunkte* Ausdrücke x_i vorliegen, so müßte ein Klient, der eine bestimmte Aktion ausführen möchte, immer nur denjenigen Manager um Erlaubnis fragen, in dessen Ausdruck die Aktion vorkommt. Auf diese Weise würde sich die Anfragelast der Klienten, die einen einzelnen Manager möglicherweise überfordern würde, auf n Manager verteilen. Kommt eine Aktion in mehreren Ausdrücken vor, so muß der Klient entsprechend mehrere Manager um Erlaubnis fragen. Je nach „Überlappungsgrad“ der Ausdrücke erreicht man aber auch in diesem Fall u. U. eine erhebliche Entlastung der einzelnen Manager im Vergleich zu einem zentralen Manager.

5.2.7.2 Schwierigkeiten

Eine wichtige Voraussetzung für den Einsatz mehrerer Interaktionsmanager ist jedoch die Möglichkeit, die Zustandsübergänge ihrer Ausdrücke bei Bedarf miteinander zu *synchronisieren*, um inkonsistente Gesamtzustände (und möglicherweise auch Verklemmungssituationen) zu vermeiden. Hat man beispielsweise zwei Manager gegeben, die die Ausdrücke $x_1 \equiv \ominus(a - b)$ und $x_2 \equiv \ominus(a - c)$ verarbeiten (vgl. Abb. 5.11), so kann bei Verwendung des dreiphasigen Koordinationsprotokolls folgendes Problem auftreten: Ein Klient will die Aktion a ausführen und stellt daher an beide Manager eine entsprechende Ask-Anfrage. Da die Aktion in beiden Graphen durchlaufen werden kann, antworten beide Manager mit Accept. Daraufhin beginnt der Klient eine Transaktion, führt die Aktion a aus, sichert die Transaktion und verschickt Exec-Bestätigungen an beide Manager. Normalerweise sollten diese nun beide den Zustandsübergang für a durchführen und Commit-Antworten an den Klienten zurücksenden. Wenn sich aber die Ankunft *einer* der beiden Exec-Nachrichten aus irgendwelchen Gründen verzögert, so kann es passieren, daß der entsprechende Manager „die Geduld verliert“, den Zustandsübergang *nicht* durchführt und eine Abort-Antwort an den Klienten schickt. Abgesehen davon, daß der Klient nun zwei widersprüchliche Antworten erhält − von einem Manager Commit, vom anderen Abort −, befinden sich nun auch die beiden Manager in *inkonsistenten Zuständen*: Im einen Graphen wurde a durchlaufen, im anderen nicht.

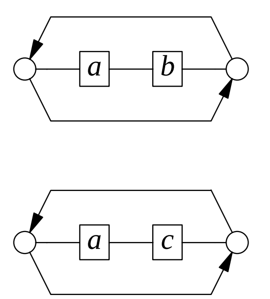


Abbildung 5.11: Gefahr inkonsistenter Zustände

Der große Vorteil des dreiphasigen Koordinationsprotokolls − die Tatsache, daß ein Manager eine vorläufige Accept-Antwort jederzeit durch eine endgültige Abort-Antwort widerrufen kann − erweist sich beim Einsatz mehrerer Manager als ebenso großer Nachteil, da man in einem System, in dem *mehrere* Beteiligte dieses Recht besitzen, keine verbindlichen Entscheidungen mehr treffen kann.

5.2.7.3 Modifiziertes Koordinationsprotokoll

Um das zuletzt beschriebene Problem zu lösen, darf ein Manager bei Ausbleiben der Exec- oder Undo-Bestätigung des Klienten die Antwort Abort nicht mehr eigenmächtig verschicken, sondern muß sich zuvor mit den übrigen von der Anfrage betroffenen Managern *abstimmen*: Sofern einer von ihnen bereits eine Bestätigung des Klienten erhalten hat, können die übrigen Manager diese Nachricht übernehmen und entsprechend reagieren. Entsprechendes gilt, wenn einer der Manager Reject geantwortet hat, weil in diesem Fall jeder der Manager früher oder später eine Undo-Bestätigung des Klienten erhalten wird. Andernfalls, d. h. wenn *alle* betroffenen Manager Accept geantwortet haben, aber *keiner* bisher eine Bestätigung des Klienten erhalten hat, können sie *gemeinsam* entscheiden, Abort-Antworten an den Klienten zu schicken. Dadurch ist zum einen gewährleistet, daß der Klient in jedem Fall *konsistente Antworten* erhält − entweder von allen Managern Commit oder von allen Abort −, und zum anderen, daß die Ausdrücke aller Manager *konsistente Zustände* besitzen − entweder führen alle Manager den Zustandsübergang durch oder sie führen ihn alle nicht durch.

Die konkreten Details dieses Abstimmungsprozesses sollen an dieser Stelle nicht weiter ausgeführt werden. Wichtig ist jedoch, daß *alle* betroffenen Interaktionsmanager eine gemeinsame, *konsistente Entscheidung* fällen. Dies bedeutet insbesondere, daß bei Ausfall eines dieser Manager weder eine gemeinsame Abort- noch eine gemeinsame Commit-Entscheidung getroffen werden kann, da der ausgefallene Manager möglicherweise bereits eine hiermit unverträgliche Antwort an den Klienten geschickt hat. (Wenn er anfangs Reject geantwortet hat, kann kein globales Commit mehr zustandekommen; hat er Accept geantwortet, die Exec-Bestätigung des Klienten erhalten und diese bereits mit Commit beantwortet, so kann kein globales Abort mehr zustandekommen.) Dies bedeutet, daß ein Ausfall eines Managers (bzw. der Verbindung zu ihm) dazu führen kann, daß weitere Manager vorübergehend handlungsunfähig werden. Im Gegensatz zu einer Ein-Manager-Architektur, bei der ein Ausfall des einzigen vorhandenen Managers zwangsläufig das ganze System lahmlegt, können beim Einsatz mehrerer Manager einige von ihnen durchaus handlungsfähig bleiben und Klienten bedienen, die keinen der momentan ausgefallenen oder blockierten Manager benötigen.

Aus der Sicht eines Klienten gestaltet sich das modifizierte dreiphasige Koordinationsprotokoll nun wie folgt (vgl. Abb. 5.7):

1. Der Klient schickt Ask-Anfragen an alle Interaktionsmanager, die die auszuführende Aktion a kennen.
2. Antwortet *einer* dieser Manager mit Reject, darf die Aktion momentan nicht ausgeführt werden. In diesem Fall beantwortet der Klient eventuelle Accept-Antworten anderer Manager mit Undo.
   Antworten *alle* betroffenen Manager mit Accept, so beginnt der Klient eine Transaktion, führt die Aktion a aus, sichert die Transaktion und verschickt Exec-Bestätigungen an alle beteiligten Manager.
3. Antwortet nun einer der Manager mit Abort, so muß die Transaktion zurückgesetzt werden; trifft jedoch eine Commit-Antwort ein, so kann die Transaktion erfolgreich abgeschlossen werden. Alle übrigen Antworten können im Prinzip ignoriert werden, da sie identisch zur ersten eingetroffenen Antwort sein werden.

5.2.7.4 Anmerkungen

Da die Antwort auf eine Wait-Anfrage auch unter normalen Umständen sehr lange ausbleiben kann, ist dieser Anfragetyp für ein Koordinationsprotokoll mit mehreren Interaktionsmanagern grundsätzlich ungeeignet. Mit Hilfe abonnierter Nachrichten ist ein Klient aber dennoch in der Lage, passiv (d. h. ohne „busy wait“) auf einen Zeitpunkt zu warten, zu dem eine bestimmte Aktion mit großer Wahrscheinlichkeit von allen betroffenen Managern akzeptiert wird (vgl. § 5.2.5).

Da das *zweiphasige* Koordinationsprotokoll im wesentlichen einem Zwei-Phasen-Commit-Protokoll entspricht (bei dem der Klient die Rolle des Commit-Koordinators spielt; vgl. Fußnote in § 5.2.3.2),

könnte es für den Einsatz mehrerer Interaktionsmanager *unverändert* übernommen werden. Allerdings sollten die Klienten in diesem Fall sehr zuverlässige Programme sein, deren Wichtigkeit mit der eines Interaktionsmanagers vergleichbar ist, damit die in § 5.2.3.2 diskutierten Probleme vernachlässigt werden können. Handelt es sich beispielsweise nicht um Arbeitslistenprogramme, sondern um zentrale Ausführungseinheiten eines oder mehrerer Workflow-Management-Systeme, so stellt das zweiphasige Protokoll aufgrund seiner einfacheren Struktur eine echte Alternative zum oben beschriebenen dreiphasigen Protokoll dar.

5.2.8 Partitionierung von Graphen

5.2.8.1 Motivation

Wollte man den Einsatz mehrerer Interaktionsmanager konkret am Beispiel der in § 2.7 entwickelten und in Abb. 2.73 zusammengefaßten Integritätsbedingungen für medizinische Untersuchungsworkflows demonstrieren (indem man jeden Graphen an einen eigenen Manager übergibt), so würde man feststellen, daß der „Überlappungsgrad“ der Graphen so hoch ist, daß der Einsatz mehrerer Manager lediglich zu einem höheren Kommunikationsaufkommen, nicht jedoch zu einer Lastverteilung zwischen den Managern führen würde. Insbesondere müßte der Manager des obersten Graphen in jede Entscheidung miteinbezogen werden, da sein Graph die Aktionen der übrigen Graphen ebenfalls enthält. Somit wäre es natürlich einfacher und effizienter, die Graphen mittels einer Kopplung zu verknüpfen und von einem einzigen Manager verwalten zu lassen.

Dies würde jedoch bedeuten, daß sämtliche Überlegungen zum Einsatz mehrerer Interaktionsmanager für dieses konkrete Anwendungsbeispiel umsonst gewesen wären und Vorteile wie verbesserte Performance oder Skalierbarkeit des Gesamtsystems möglicherweise nur theoretisch vorhandene Optionen darstellen. Außerdem würde ein zentralistischer Interaktionsmanager, der im konkreten Beispiel *alle* in einer Klinik anfallenden Aktivitäten vom Typ Patient vorbereiten, Patient aufklären, Patient abrufen und Untersuchung durchführen verfolgen und kontrollieren müßte, sämtliche Bemühungen zur Verbesserung des Durchsatzes auf WfMS-Seite − wie z. B. der Einsatz replizierter Workflow-Ausführungseinheiten oder eine geschickte Partitionierung von Workflows [Bauer97, Bauer98, Bauer99] − mit einem Schlag zunichte machen.

Erfreulicherweise ist die Situation bei genauerer Betrachtung aber doch nicht so aussichtslos, wie sie auf den ersten Blick erscheinen mag, und tatsächlich wird es in diesem Anwendungsbeispiel sogar möglich sein, *beliebig viele* Interaktionsmanager einzusetzen, von denen bei jeder Aktionsausführung maximal zwei um Erlaubnis gefragt werden müssen.

5.2.8.2 Prinzip

Um dies einzusehen, betrachte man exemplarisch den Graphen in Abb. 5.12, bei dem es sich um eine Für-alle-Verzweigung $\underset{p}{\circledcirc}\, y$ mit einem hier nicht näher interessierenden Rumpf y handelt. Wie in § 2.3.4.4 und § 3.3.3.1 erläutert, stellt ein solcher Graph anschaulich eine Sowohl-als-auch-Verzweigung $\underset{\omega\in\Omega}{\circledcirc} y_p^\omega$ mit unendlich vielen Zweigen y_p^ω dar, die man erhält, wenn man in allen Aktionen des Rumpfs y den Parameter p jeweils durch einen Wert $\omega \in \Omega$ ersetzt (vgl. Abb. 5.13). Zerlegt man nun die Menge Ω aller möglichen Parameterwerte in beliebig viele paarweise disjunkte Teilmengen $\Omega_1, \ldots, \Omega_n$ (die ebenso wie Ω unendliche Kardinalität besitzen sollen), so kann die Für-alle-Verzweigung $x \equiv \underset{\omega\in\Omega}{\circledcirc} y_p^\omega$ auch als Sowohl-als-auch-Verzweigung $x_1 \circledcirc \ldots \circledcirc x_n$ von n unendlichen Sowohl-als-auch-Verzweigungen $x_i \equiv \underset{\omega\in\Omega_i}{\circledcirc} y_p^\omega$ $(i = 1, \ldots, n)$ aufgefaßt werden.[8] Da die Mengen Ω_i paarweise disjunkt sind, sind offensichtlich auch die Graphen x_i paarweise disjunkt, d. h. eine Aktion wie z. B. Patient p für Untersuchung u vorbereiten kommt abhängig vom konkreten Wert des Parameters p in

[8] Formal bedeutet das, daß für unendliche Verschränkungen (vgl. § 3.2.2.4) das Assoziativgesetz $\underset{\omega\in\Omega}{\otimes} U_\omega = \overset{n}{\underset{i=1}{\otimes}} \underset{\omega\in\Omega_i}{\otimes} U_\omega$ gilt.

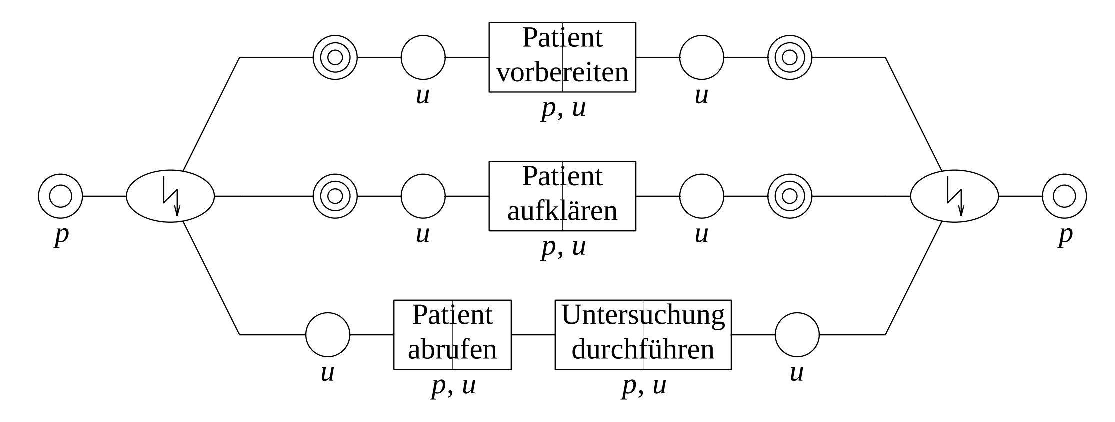


Abbildung 5.12: Integritätsbedingung für Patienten

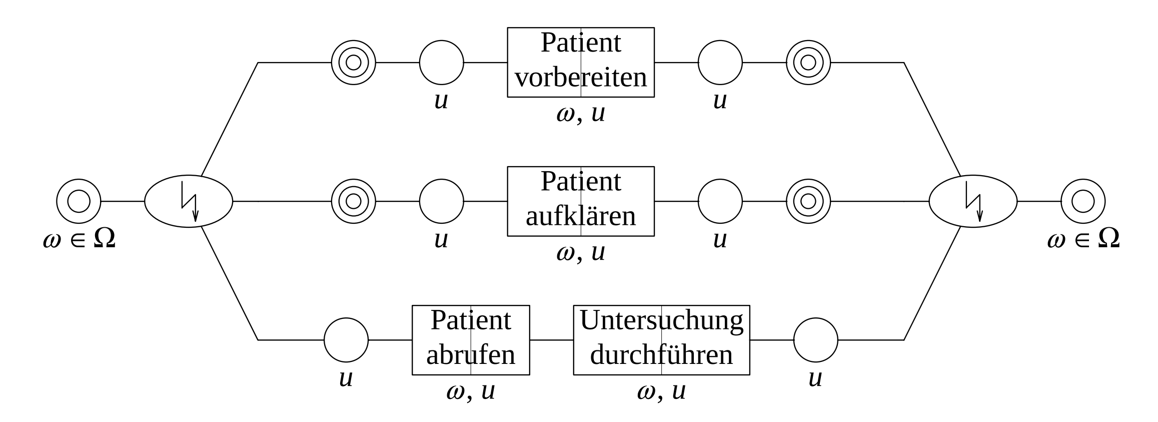


Abbildung 5.13: Für-alle-Verzweigung als unendliche Sowohl-als-auch-Verzweigung

genau *einem* Graphen x_i vor. Daraus folgt gemäß § 3.4.5.2, daß die Sowohl-als-auch-Verzweigung $x_1 \circledcirc \dots \circledcirc x_n$ äquivalent zu einer Kopplung $x_1 \circ \dots \circ x_n$ ist (vgl. Abb. 5.14).

Dies wiederum impliziert, daß man jeden der Graphen x_i von einem eigenen Interaktionsmanager verwalten lassen könnte und das Resultat äquivalent zur ursprünglichen Für-alle-Verzweigung $x \equiv \underset{p}{\circledcirc}\, y$ wäre. Da die Graphen x_i außerdem paarweise disjunkt sind, wäre eine solche Verteilung optimal, da für eine Aktionsausführung immer nur *ein* Manager um Erlaubnis gefragt werden müßte.

5.2.8.3 Realisierung

Das einzige verbleibende Problem bei dieser *Partitionierung* des Graphen x ist die Frage, wie man einen Teilgraphen $x_i \equiv \underset{\omega \in \Omega_i}{\circledcirc}\, y_p^\omega$ konkret an einen Interaktionsmanager übergibt und wie dieser ihn implementieren soll. Wie es scheint, ist die in Kapitel 4 entwickelte operationale Semantik und Implementierung von Interaktionsausdrücken nur in der Lage, *vollständige* unendliche Graphen $x \equiv \underset{p}{\circledcirc}\, y \equiv \underset{\omega \in \Omega}{\circledcirc}\, y_p^\omega$ zu verarbeiten, bei denen ω die *gesamte* Menge Ω durchläuft. Bei genauerer Betrachtung stellt man jedoch fest, daß der konkrete Inhalt der Menge Ω weder für die operationale Semantik noch für die tatsächliche Implementierung irgendeine Rolle spielt, d. h. daß die Graphen x und x_i, die sich nur durch die Menge Ω bzw. Ω_i unterscheiden, vollkommen identisch implementiert werden können! (Welcher Ausdruck tatsächlich implementiert wird, hängt letztlich nur davon ab, welche Parameterwerte in den konkret ausgeführten Aktionen enthalten sind.)

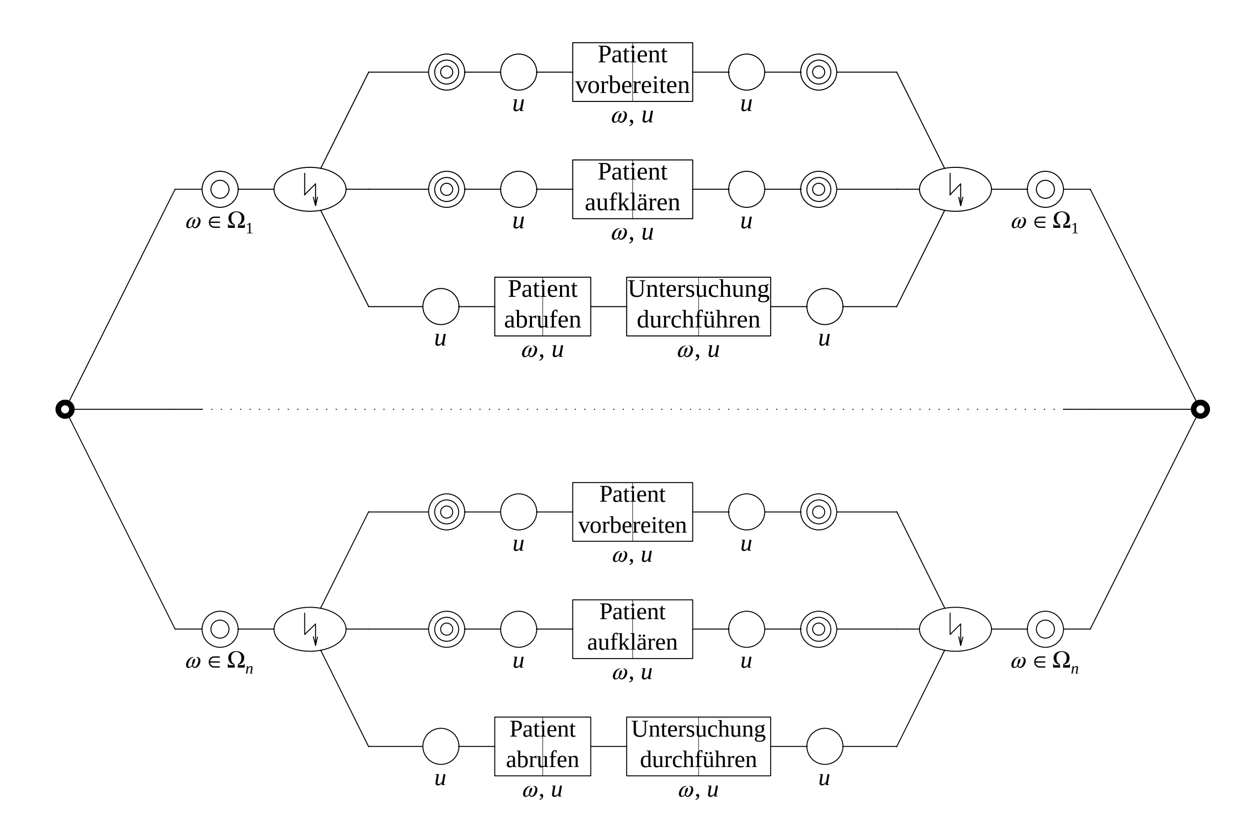


Abbildung 5.14: Für-alle-Verzweigung als Kopplung unendlicher Sowohl-als-auch-Verzweigungen

Konkret bedeutet dies, daß man anstelle der Teilgraphen x_i an jeden Manager den *vollständigen* Graphen x aus Abb. 5.12 übergibt und lediglich die Menge der Aktionen einschränkt, für die er „zuständig" ist. Hierbei muß sichergestellt werden, daß Aktionen mit demselben Wert des Parameters p immer von ein und demselben Manager bearbeitet werden. Beispielsweise könnten alle Klienten eine bestimmte Hashfunktion verwenden, die jedem möglichen Wert für p eindeutig eine Managernummer $i \in \{ 1, \ldots, n \}$ zuordnet.

Zur Veranschaulichung dieser, auf den ersten Blick möglicherweise überraschenden Erkenntnis, stelle man sich beispielsweise zwei verschiedene Finanzämter vor. Im einen Amt werden alle Steuererklärungen von einem einzigen Beamten bearbeitet, während die Arbeit im anderen Amt auf mehrere Beamte verteilt wird. Beispielsweise ist dort ein Beamter für die Anfangsbuchstaben A−H, ein zweiter für I−P und ein dritter für Q−Z zuständig. Vergleicht man die Arbeit dieser drei Beamten mit der des einen Beamten im ersten Finanzamt, so stellt man fest, daß ihre Tätigkeiten vollkommen identisch sind und sich durch nichts unterscheiden: Jeder Beamte nimmt Steuererklärungen entgegen, prüft sie auf Vollständigkeit und Richtigkeit usw. Daß der eine Beamte Steuererklärungen mit beliebigen Anfangsbuchstaben bearbeitet, während die anderen nur eine Teilmenge des Alphabets erhalten, ist vollkommen irrelevant.

Wesentlich für eine korrekte Bearbeitung der Steuererklärungen ist jedoch, daß jeder Steuerpflichtige seine Erklärung jedes Jahr an denselben Beamten richtet, da nur dieser die Unterlagen der vergangenen Jahre besitzt. Würde jemand den Beamten wechseln, so würde der neue Beamte in seinem Schrank keine Akte für diese Person finden und daher (wenn er nicht auf den Anfangsbuchstaben achtet) fälschlicherweise eine neue anlegen. Auf diese Weise könnte die Person beispielsweise zu Unrecht gewisse Steuervergünstigungen mehrmals erhalten.

Zurückübertragen auf das Szenario der n Interaktionsmanager, die alle denselben Graphen x verarbeiten, könnte das z. B. bedeuten, daß sich ein Klient quasi gleichzeitig von zwei verschiedenen Managern die Erlaubnis zum Starten der Aktivitäten Patient Meier für Untersuchung sono vorbereiten

und Patient Meier für Untersuchung endo aufklären einholen könnte, die er von einem einzigen Manager so nicht erhalten würde.

5.2.8.4 Verallgemeinerung

Ebenso wie der Graph aus Abb. 5.12, lassen sich grundsätzlich beliebige Für-alle-Verzweigungen $\underset{p}{\circledcirc}\, y$ anhand der Werte des Parameters p partitionieren, sofern ihr Rumpf y gewisse Voraussetzungen erfüllt, die bei realistischen Graphen normalerweise gegeben sind. Eine hinreichende Bedingung ist z. B., daß y *vollständig und homogen quantifiziert* ist, d. h. daß jede Aktion a von y den Parameter p enthält und − falls a in y mehrmals auftritt − daß p in der Argumentliste von a immer an derselben Position steht. In diesem Fall sind die logisch gebildeten Teilgraphen $x_i \equiv \underset{\omega \in \Omega_i}{\circledcirc} y_p^\omega$ paarweise disjunkt (vgl. § 4.7.2.2), so daß die Sowohl-als-auch-Verzweigung $x_1 \circledcirc \ldots \circledcirc x_n$ äquivalent zu der Kopplung $x_1 \circ \ldots \circ x_n$ ist.

Wendet man das Prinzip der Partitionierung auch auf den Graphen in Abb. 5.15 an, der ebenso wie der zuvor betrachtete Graph x eine Integritätsbedingung $x' \equiv \underset{p}{\circledcirc}\, y'$ für alle Patienten p formuliert, so erhält man entsprechend n logische Teilgraphen $x'_1, \ldots, x'_n$, für die gilt: $x' = x'_1 \circ \ldots \circ x'_n$. Aufgrund des Assoziativ- und Kommutativgesetzes (vgl. § 3.4.7.2) kann die Kopplung $x \circ x'$ der beiden Graphen x und x' daher wie folgt partitioniert werden:

$$x \circ x' = (x_1 \circ \ldots \circ x_n) \circ (x'_1 \circ \ldots \circ x'_n) = (x_1 \circ x'_1) \circ \ldots \circ (x_n \circ x'_n).$$

Da nun jeder Teilgraph $x_i \circ x'_i$ wieder identisch wie der vollständige Graph $x \circ x'$ implementiert werden kann, läßt sich das Prinzip der Partitionierung auf Kopplungen von Für-alle-Verzweigungen verallgemeinern, sofern diese anhand desselben Parameters verzweigen.

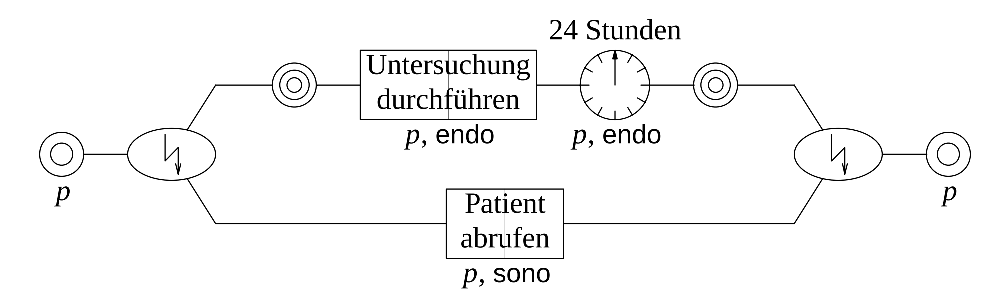


Abbildung 5.15: Mindestabstand zwischen Endoskopie und Sonographie

5.2.8.5 Anwendung

Konkret bedeutet das, daß man zur Implementierung der Bedingungen x und x' n Interaktionsmanager $N_1, \ldots, N_n$ einsetzen kann, die alle den Graphen $x \circ x'$ verarbeiten, aber jeweils nur für eine bestimmte Teilmenge P_i aller Patienten p zuständig sind. Analog können auch die beiden Graphen in Abb. 5.16 durch den Einsatz von m Managern $M_1, \ldots, M_m$ implementiert werden, die alle die Kopplung dieser beiden Graphen verarbeiten und jeweils für eine bestimmte Teilmenge U_j aller Untersuchungen u zuständig sind. Somit wird der gesamte zweidimensionale Parameterraum aller Patienten-Untersuchungs-Kombinationen (p, u) schachbrettartig in insgesamt $m \cdot n$ Felder $P_i \times U_j$ unterteilt, von denen jedes einerseits von einem Manager N_i und andererseits einem Manager M_j „kontrolliert“ wird (vgl. Abb. 5.17).

Will ein Klient nun eine Aktion für einen bestimmten Patienten p und eine bestimmte Untersuchung u ausführen, so muß er zunächst mit Hilfe zweier Hashfunktionen o. ä. die beiden Indizes i

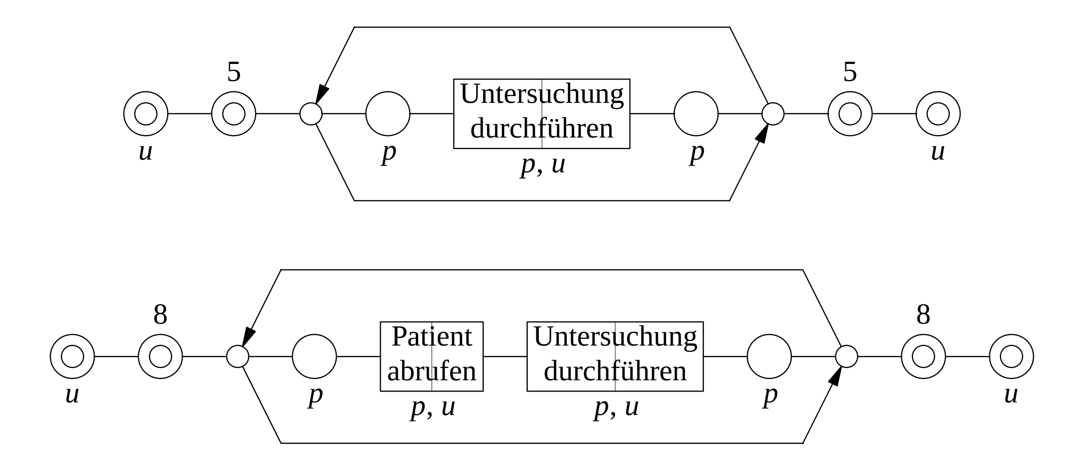


Abbildung 5.16: Bedingungen für Untersuchungsstellen

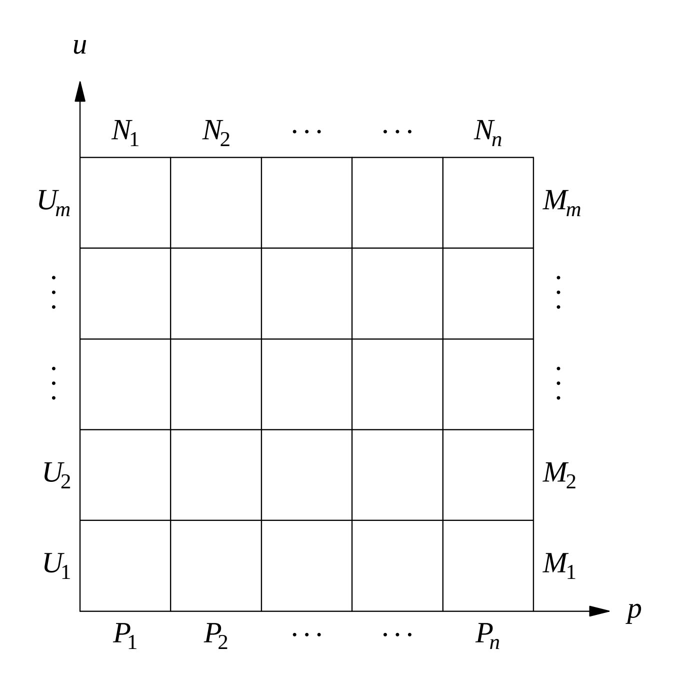


Abbildung 5.17: Aufteilung des zweidimensionalen Parameterraums

und j bestimmen, für die $p \in P_i$ und $u \in U_j$ gilt. Anschließend kann er unter Verwendung des zweiphasigen oder des modifizierten dreiphasigen Koordinationsprotokolls die beiden Manager N_i und M_j um die entsprechende Ausführungserlaubnis bitten.

Der letzte verbleibende Graph aus Abb. 2.73, bei dem es sich ausnahmsweise nicht um eine Für-alle-Verzweigung handelt, könnte entweder von einem eigenen Manager verwaltet werden oder aber bei demjenigen Manager M_j mitangekoppelt werden, der für den Parameterwert u = sono zuständig ist. Auf diese Weise hat man tatsächlich das in § 5.2.8.1 formulierte Ziel erreicht, daß die Menge der Graphen aus Abb. 2.73 mit beliebig vielen Interaktionsmanagern implementiert werden kann, von denen pro Aktionsausführung maximal zwei um Erlaubnis gefragt werden müssen. (Für Aktionen, die in keinem der Graphen vorkommen, muß natürlich keiner der Manager gefragt werden.) Dies bedeutet, daß das resultierende Gesamtsystem im Prinzip beliebig skalierbar ist.

5.2.9 Implementierung eines Interaktionsmanagers

Die Abbildungen 5.18 bis 5.21, die im folgenden näher erläutert werden, zeigen die Kernteile der Implementierung eines Interaktionsmanagers, der über das dreiphasige Koordinationsprotokoll mit seinen Klienten kommuniziert. Für ein detailliertes Verständnis des Programmcodes sind die Ausführungen zur Programmiersprache `CH` in § 4.3.3, § 4.4.2 und Anhang A sowie Kenntnisse der Implementierung in § 4.6 erforderlich. Zum Verständnis der wesentlichen Konzepte sollten jedoch die nachfolgenden Erläuterungen sowie die im Programmcode enthaltenen Kommentare genügen. Ebenso wie Kapitel 4, kann der vorliegende Abschnitt auch komplett übersprungen werden.

```
// Nachrichtentypen (message categories).
type MsgCat;
inst MsgCat { Ask, Wait };              // Anfragen.
inst MsgCat { Accept, Reject, Unknown }; // (Vorläufige) Antworten.
inst MsgCat { Exec, Undo };             // Bestätigungen.
inst MsgCat { Commit, Abort };          // Endgültige Antworten.
inst MsgCat { Subscribe, Permit, Forbid, Cancel }; // Abonnement.

// Nachricht.
type Msg;
attr cat: Msg -> MsgCat;        // Typ (Kategorie) der Nachricht.
attr action: Msg -> Action;     // Angefragte Aktion.

// Gloable Variablen.
Expr x;                         // Zu verarbeitender Ausdruck.
State s;                        // Aktueller Zustand des Ausdrucks.
Seq(Msg) queue = seq();         // Liste unbeantworteter Wait-Anfragen.
Seq(Msg) sub = seq();           // Liste von Abonnenten.
```

Abbildung 5.18: Implementierung eines Interaktionsmanagers (Teil 1: Deklarationen)

5.2.9.1 Hauptprogramm (Abb. 5.19)

Ebenso wie das in § 4.6.5 beschriebene Programm, erhält auch ein Interaktionsmanager einen Interaktionsausdruck *x*, den er verarbeiten soll, als Kommandoargument `args[1]`. Er wandelt ihn mit Hilfe der Parserfunktion `parse:expr()` in einen Operatorbaum `x` um und bestimmt dessen initialen Zustand `s = init(x)`. Anschließend verarbeitet er in einer Endlosschleife Anfragen von Klienten, die mit Hilfe einer hier nicht näher erläuterten Funktion `receive()` entgegengenommen und in Objekte vom Typ `Msg` umgewandelt werden. Neben den in Abb. 5.18 vereinbarten Attributen `cat` (Typ bzw. Kategorie der Nachricht) und `action` (angefragte Aktion) enthält ein solches Objekt auch den Absender der Nachricht, d. h. den Klienten, an den eine eventuelle Antwortnachricht zurückgesandt wird.

Mit Hilfe der Funktion `contains()` (vgl. § 4.6.2.2) wird zunächst überprüft, ob der Ausdruck `x` die angefragte Aktion `m/action` enthält. Ist dies nicht der Fall, wird mit Hilfe der Funktion `send()` entweder die Antwort `Unknown` (bei Anfragen vom Typ `Ask` oder `Wait`) oder `Permit` (bei Anfragen vom Typ `Subscribe`) zurückgesandt, während eine `Cancel`-Nachricht in diesem Fall vollkommen unbeantwortet bleibt. Als Gegenstück zu `receive()` erhält die Funktion `send()` ein Objekt vom Typ `Msg` als Argument (hier konkret das Objekt `m`, an dessen Attribut `cat` der Wert `Unknown` oder `Permit` zugewiesen wurde) und wandelt es in eine entsprechende Nachricht um, die an den oben erwähnten Absender von `m` zurückgesandt wird.

Nach Ausschluß dieses Sonderfalls wird die Anfrage `m` abhängig vom Nachrichtentyp `m/cat` jeweils unterschiedlich verarbeitet.

```
begin {
    // Ausdruck in einen Operatorbaum transformieren
    // und seinen initialen Zustand bestimmen.
    x = parse:expr(args[1]);
    s = init(x);

    // Nachrichten von Klienten vearbeiten.
    loop {
        Msg m = receive();

        if (!contains(x, m/action)) {
            // Je nach Kategorie entweder Unknown,
            // Permit oder nichts antworten.
            if (m/cat == Ask || m/cat == Wait) send(m(cat = Unknown));
            else if (m/cat == Subscribe) send(m(cat = Permit));
        }
        else if (m/cat == Ask) {
            // Entweder Accept oder Reject antworten
            // und ggf. weitere Protokollschritte durchlaufen.
            if (reply(m, Accept, Reject) == 2) complete();
        }
        else if (m/cat == Wait) {
            // Entweder Accept oder nichts antworten
            // und ggf. weitere Protokollschritte durchlaufen.
            // Ggf. Anfrage in Warteliste stellen.
            switch (reply(m, Accept, nil)) {
            case 2: complete(); break;
            case 0: queue += m; break;
            }
        }
        else if (m/cat == Subscribe) {
            // Entweder Permit oder nichts antworten.
            // Anfrage in Abonnentenliste eintragen.
            if (reply(m, Permit, nil) == 0) m(cat = Forbid);
            sub += m;
        }
        else if (m/cat == Cancel) {
            // Nachricht aus Abonnentenliste entfernen.
            sub -= m;
        }
    }
}
```

Abbildung 5.19: Implementierung eines Interaktionsmanagers (Teil 2: Hauptprogramm)

5.2.9.2 Hilfsfunktion `reply()` (Abb. 5.20)

Anfragen vom Typ `Ask` werden von der zentralen Hilfsfunktion `reply()` verarbeitet, die in diesem Fall mit den Argumenten `yes` = `Accept` und `no` = `Reject` aufgerufen wird.

Um festzustellen, ob die Aktion `m/action` im aktuellen Zustand `s` des Ausdrucks zulässig ist, wird zunächst ein „tentativer" Zustandsübergang `trans()` ausgeführt. Falls dieser einen gültigen Folgezustand `s_` liefert, wird eine Antwort vom Typ `yes` (im konkreten Fall `Accept`) zurückgesandt und auf eine Bestätigung des Klienten gewartet. Diese wird, wie jede andere Nachricht auch, mit Hilfe der

```
// Anfrage m mit Antwort vom Typ yes (Accept oder Permit)
// bzw. no (Reject oder Forbid) beantworten,
// je nachdem ob der Zustandsübergang für m/action gelingt oder nicht.
// Falls Accept geantwortet wird, Bestätigung des Klienten abwarten,
// ggf. Zustandsübergang tatsächlich durchführen und Commit antworten;
// bei Zeitüberschreitung Abort antworten.
int reply(Msg m, MsgCat yes, MsgCat no) {
    // Zustandsübergang tentativ ausführen.
    if (State s_ = trans(s, m/action)) {
        if (yes) send(m(cat = yes));
        if (yes == Accept) {
            // Auf Bestätigung des Klienten warten.
            m = receive(m);
            if (m/cat == Exec) {
                // Zustandsübergang tatsächlich durchführen
                // und Commit antworten.
                s = s_;
                send(m(cat = Commit));
                return 2;
            }
            else if (m/cat == Undo) {
                // Nichts.
            }
            else {
                // Zeitüberschreitung: Abort antworten.
                send(m(cat = Abort));
            }
        }
        return 1;
    }
    else {
        if (no) send(m(cat = no));
        return 0;
    }
}
```

Abbildung 5.20: Implementierung eines Interaktionsmanagers (Teil 3: Hilfsfunktion `reply()`)

Funktion `receive()` empfangen und dekodiert. Im Gegensatz zum Aufruf in Abb. 5.19, bei dem `receive()` einfach die nächste anstehende Nachricht liefert, erhält die Funktion hier die aktuelle Nachricht `m` als Argument, um anzuzeigen, daß explizit auf eine Nachricht gewartet werden soll, die vom selben Absender wie `m` stammt und sich auf dieselbe Aktion `m/action` bezieht. Falls eine solche Nachricht rechtzeitig eintrifft, muß sie vom Typ `Exec` oder `Undo` sein. Im ersten Fall wird nun der „tentativ" ausgeführte Zustandsübergang tatsächlich durchgeführt, indem der aktuelle Zustand `s` durch den Folgezustand `s_` ersetzt wird, und eine abschließende `Commit`-Antwort an den Klienten geschickt. Im Falle einer `Undo`-Bestätigung des Klienten bleibt der aktuelle Zustand `s` unverändert, d. h. der Folgezustand `s_` wird verworfen. Trifft die erwartete Bestätigung des Klienten nicht rechtzeitig ein, liefert `receive()` die als Argument übergebene Nachricht `m` zurück, deren Kategorie `m/cat` ver-

```
// Zustandsübergang abschließen, d. h. Warteliste überprüfen
// und Abonnenten über Statusänderungen informieren.
proc complete() {
    Msg m;

    // Warteliste durchlaufen.
    // Wenn ein Zustandsübergang erfolgreich ist, Eintrag entfernen.
    // Wenn ein tatsächlicher Zustandsübergang durchgeführt wurde,
    // von vorne beginnen.
    forall (m << queue) {
        switch (reply(m, Accept, nil)) {
        case 1: delete; break;
        case 2: delete; reset;
        }
    }

    // Abonenntenliste durchlaufen.
    // Wenn sich der Status einer Aktion geändert hat,
    // entsprechende Nachricht verschicken.
    forall (m << sub) {
        if (m/cat == Permit) reply(m, nil, Forbid);
        else reply(m, Permit, nil);
    }
}
```

Abbildung 5.21: Implementierung eines Interaktionsmanagers (Teil 4: Hilfsfunktion `complete()`)

schieden von `Exec` und `Undo` ist. In diesem Fall sendet der Manager eine `Abort`-Antwort an den Klienten.[9]

Ist der Folgezustand `s_` ungültig, so ist die angefragte Aktion momentan nicht zulässig, was durch Zurücksenden einer Antwort vom Typ `no` (im konkreten Fall `Reject`) angezeigt wird.

Der Anfragetyp `Wait` wird sehr ähnlich behandelt, allerdings wird durch den Aufruf von `reply()` mit `no = nil` angezeigt, daß im Fall eines ungültigen Folgezustands `s_` *keine* Antwort an den Klienten geschickt werden soll. Stattdessen wird im Hauptprogramm dafür gesorgt, daß die Anfrage `m` in diesem Fall (der durch den Rückgabewert `0` der Funktion `reply()` angezeigt wird) in die Liste `queue` aller bisher unbeantworteten `Wait`-Anfragen eingetragen wird (vgl. Abb. 5.19).

5.2.9.3 Verwaltung von Abonnenten (Abb. 5.19)

Eine Anfrage vom Typ `Subscribe` wird in die Liste `sub` aller Abonnenten (engl. subscribers) eingetragen, aus der sie später durch eine entsprechende `Cancel`-Nachricht wieder entfernt werden kann. Außerdem wird bei `Subscribe` — wiederum mit Hilfe der Funktion `reply()` — überprüft, ob die angefragte Aktion momentan zulässig wäre. Ist dies der Fall, schickt `reply()` eine `Permit`-Nachricht an den Klienten zurück, läßt den aktuellen Zustand `s` jedoch unverändert. Um später feststellen zu können, ob sich der Status der Aktion `m/action` von unzulässig auf zulässig oder umgekehrt geändert hat, wird der aktuelle Status in Form des Nachrichtentyps `Permit` oder `Forbid` im Attribut `cat` der Nachricht `m` hinterlegt. Im Fall von `Permit` wurde die entsprechende Zuweisung bereits in der Funktion `reply()` durchgeführt, während sie andernfalls im Hauptprogramm vorgenommen wird.

[9] Wenn die Bestätigung des Klienten zu einem späteren Zeitpunkt eintrifft, wird sie durch einen Aufruf von `receive()` im Hauptprogramm entgegengenommen. Da die Nachrichtentypen `Exec` und `Undo` in der darauffolgenden Fallunterscheidung nicht auftreten, wird die Nachricht dort einfach ignoriert.

5.2.9.4 Hilfsfunktion `complete()` (Abb. 5.21)

Wie in den vorangegangenen Teilabschnitten erwähnt, verwaltet der Interaktionsmanager zum einen eine Liste `queue` aller `Wait`-Anfragen, die noch nicht mit `Accept` beantwortet werden konnten, und zum anderen eine Liste `sub` von Abonnenten, die mittels `Permit`- und `Forbid`-Nachrichten über Statusänderungen bestimmter Aktionen informiert werden wollen.

Da eine Veränderung des aktuellen Zustands `s` Auswirkungen auf den Status der dort gespeicherten Aktionen haben kann, wird im Anschluß an einen tatsächlichen Zustandsübergang (der durch den Rückgabewert `2` der Funktion `reply()` angezeigt wird) jeweils die Hilfsfunktion `complete()` aufgerufen (vgl. Abb. 5.19). Diese Funktion überprüft zunächst − wiederum mit Hilfe von `reply()` −, ob es bislang unbeantwortete `Wait`-Anfragen in der Liste `queue` gibt, die jetzt positiv mit `Accept` beantwortet werden können. Ist dies der Fall (Rückgabewert `1` oder `2`), wird die Anfrage mit Hilfe der Anweisung `delete` aus der Warteliste `queue` entfernt.[10] Wurde darüber hinaus ein tatsächlicher Zustandsübergang durchgeführt (Rückgabewert `2`) − was nur der Fall ist, wenn rechtzeitig eine `Exec`-Bestätigung des Klienten eingetroffen ist −, wird die `forall`-Schleife durch die Anweisung `reset` an den Anfang der Liste `queue` zurückgesetzt. Dies wird so lange wiederholt, bis die Liste entweder leer ist oder einmal komplett durchlaufen wurde, ohne daß ein weiterer Zustandsübergang durchgeführt werden konnte.

Nachdem auf diese Weise möglichst viele zurückgestellte Anfragen beantwortet wurden, werden anschließend − erneut mit Hilfe von `reply()` − alle Abonnenten der Liste `sub` informiert, für deren Aktionen sich durch die vorangegangenen Zustandsübergänge eine Statusänderung ergeben hat. Hierbei wird die oben erwähnte Tatsache ausgenutzt, daß die Kategorie `m/cat` einer in der Liste `sub` gespeicherten Nachricht `m` jeweils den aktuellen Status der Aktion `m/action`, d. h. den Typ der zuletzt für diese Aktion versandten Mitteilung, enthält. (Wurde noch keine Mitteilung versandt, gilt `m/cat == Forbid`; vgl. § 5.2.9.3.) Da das Attribut `cat` einer Nachricht `m` in der Funktion `reply()` jeweils aktualisiert wird, wenn eine Nachricht an den Absender von `m` verschickt wird, wird diese Invariante jederzeit aufrechterhalten.

5.2.10 Wiederanlauf nach Systemausfällen

Ein wesentlicher Mangel der Implementierung in § 5.2.9 besteht darin, daß sämtliche Datenstrukturen ausschließlich im Hauptspeicher gehalten werden, wo sie im Fall eines Programm- oder Rechnerabsturzes unwiederbringlich verloren gehen. Dies betrifft sowohl den aktuellen Zustand `s` des Ausdrucks `x`, ohne den eine Beantwortung von Klientenanfragen nicht möglich ist, als auch die Warteliste `queue` und die Abonnentenliste `sub`, die für einen korrekten Betrieb eines Interaktionsmanagers ebenfalls unentbehrlich sind. Darüber hinaus können bei einem Systemausfall auch Nachrichten verloren gehen, die sich im Eingangspuffer des Interaktionsmanagers befinden und noch nicht verarbeitet wurden oder die vom Manager bereits verschickt, von der Netzwerk-Software aber noch nicht endgültig zugestellt wurden.

5.2.10.1 Schutz vor Nachrichtenverlusten

Der Gefahr von Nachrichtenverlusten kann man grundsätzlich durch die Verwendung *persistenter Nachrichtenkanäle* (engl. persistent message queues) [Bernstein90, Gray93, Mohan94] begegnen, die entweder mit Hilfe eines konventionellen Datenbanksystems implementiert werden können oder durch den Einsatz geeigneter Middleware-Komponenten (z. B. Transaktionsmonitore) ohnehin zur Verfügung stehen. Im einfachsten Fall besitzt sowohl ein Interaktionsmanager als auch jeder Klient einen persistenten *Eingangskanal*, aus dem er Nachrichten entgegennimmt, die an ihn adressiert sind. Um eine Nachricht zu verschicken, muß man diese direkt in den Eingangskanal des Empfängers schreiben (vgl. Abb. 5.22). Da ein Empfänger (bzw. sein Eingangskanal) aber vorübergehend nicht er-

[10] `delete` entfernt das aktuelle Iterationselement der umgebenden `forall`-Schleife aus der zugehörigen Sequenz, ohne die Iteration zu „stören“ oder in einen undefinierten Zustand zu versetzen. Im nächsten Iterationsschritt wird daher wie gewohnt das nächste Element der Sequenz `queue` bearbeitet.

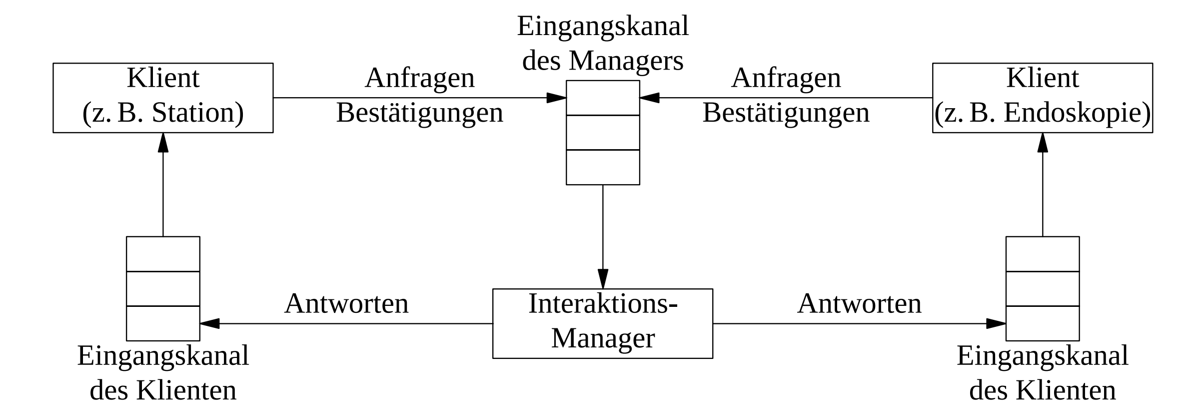


Abbildung 5.22: Verwendung persistenter Nachrichtenkanäle

reichbar sein kann, ist es zumindest für einen Interaktionsmanager sinnvoll, einen zusätzlichen *Ausgangskanal* zu besitzen, in dem er Nachrichten zwischenspeichern kann, die nicht sofort zugestellt werden können (vgl. Abb. 5.23). Auf diese Weise ist sichergestellt, daß ein Manager durch das Versenden einer Nachricht nicht blockiert werden kann.

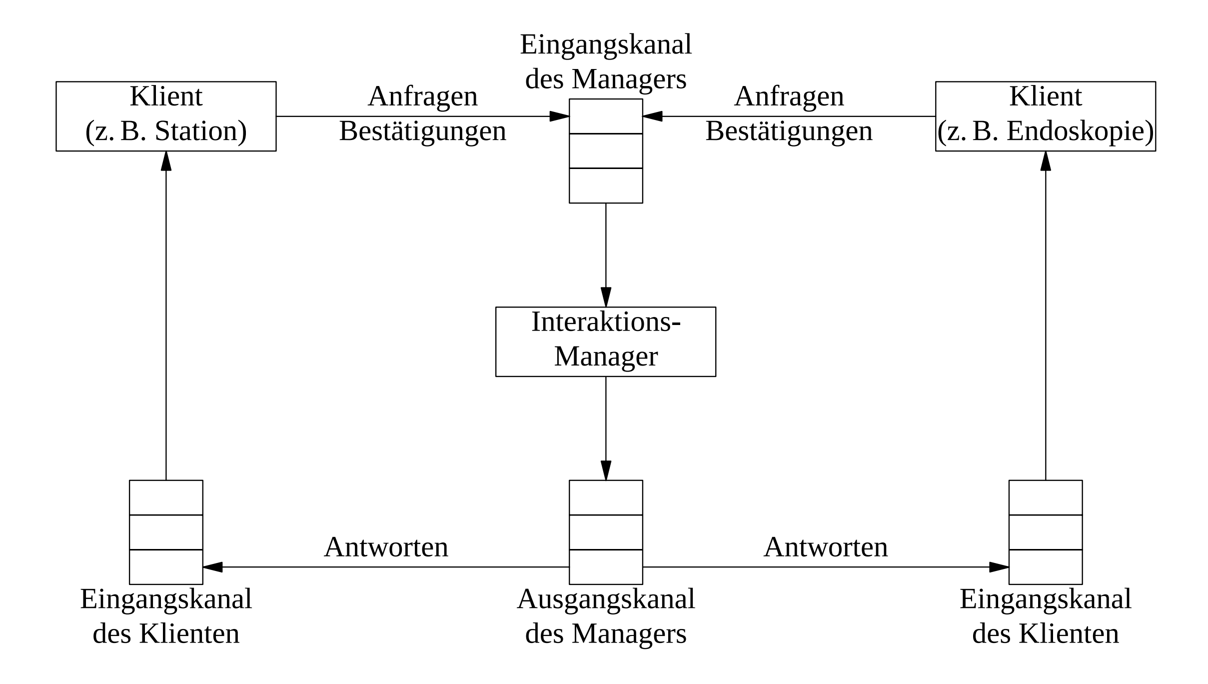


Abbildung 5.23: Zusätzlicher Ausgangskanal des Interaktionsmanagers

5.2.10.2 Schutz vor Datenverlusten

Um die kritischen Datenstrukturen `s`, `queue` und `sub` nach einem Systemausfall wiederherstellen zu können, gibt es prinzipiell verschiedene Möglichkeiten:

1. Die Datenstrukturen werden nicht im Hauptspeicher, sondern in einer *Datenbank* gehalten. Sämtliche Änderungsoperationen, die durch das Verarbeiten einer Nachricht ausgelöst werden, finden innerhalb einer *Transaktion* statt, die auch das Entnehmen der Nachricht aus dem Eingangskanal des

Managers (d. h. den Aufruf der Funktion `receive()`) und das eventuelle Versenden einer Antwortnachricht (d. h. einen Aufruf der Funktion `send()`, durch den die Nachricht entweder direkt zugestellt oder im Ausgangskanal des Managers abgelegt wird) umfaßt. Auf diese Weise ist sichergestellt, daß die Teilschritte (1) Entgegennahme einer Nachricht, (2) Verarbeitung der Nachricht und (3) Versenden einer Antwortnachricht entweder alle ausgeführt oder aber alle nicht ausgeführt werden. Nach einem Systemabsturz sind keinerlei besondere Maßnahmen erforderlich, da Datenstrukturen und Nachrichtenkanäle zu jedem Zeitpunkt konsistent sind.

2. Die Datenstrukturen werden zwar im Hauptspeicher gehalten und manipuliert, sämtliche Änderungsoperationen werden jedoch *zuverlässig protokolliert*. Auch hierfür ist es sinnvoll, die Entgegennahme einer Nachricht, das Protokollieren der zugehörigen Änderungsoperationen und das eventuelle Versenden einer Antwortnachricht jeweils in einer Transaktion zusammenzufassen. Beim Wiederanlauf nach einem Systemabsturz startet der Interaktionsmanager mit dem initialen Zustand `s` des Ausdrucks `x` und leeren Listen `queue` und `sub`; anschließend *wiederholt* er alle zuvor protokollierten Änderungsoperationen und erreicht so den letzten konsistenten Zustand vor dem Ausfall.

3. Man protokolliert nicht die einzelnen Änderungsoperationen auf den Datenstrukturen, sondern lediglich die eingehenden *Nachrichten*, die die Operationen ausgelöst haben. Beim Wiederanlauf werden diese Nachrichten erneut gelesen und die zugehörigen Operationen ausgeführt, fast so, als ob es sich um gewöhnliche Nachrichten von Klienten handeln würde. Der einzige Unterschied besteht darin, daß während dieser Wiederanlaufphase keine Antwortnachrichten verschickt werden, weil diese zuvor bereits zuverlässig versandt wurden.

4. Da die eingehenden Nachrichten im Eingangskanal des Managers ohnehin zuverlässig gespeichert werden, kann man auf eine Protokollierung sogar vollständig verzichten, wenn man sicherstellt, daß mit `receive()` entgegengenommene Nachrichten nicht wirklich aus dem Eingangskanal entfernt, sondern lediglich als verarbeitet markiert werden.

Vergleicht man diese Alternativen bezüglich des Zusatzaufwands, den sie im *normalen Betrieb* eines Interaktionsmanagers erfordern, so ist Variante 4 zweifellos optimal. Sie besitzt jedoch − ähnlich wie auch die Varianten 2 und 3 − den Nachteil, daß der *Wiederanlauf* des Interaktionsmanagers nach einem Systemausfall umso länger dauert, je größer die Anzahl der im Eingangskanal gespeicherten Nachrichten ist. Da diese Zahl im normalen Betrieb permanent wächst, erreicht man früher oder später einen Zustand, in dem ein Wiederanlauf des Managers in akzeptabler Zeit nicht mehr möglich ist. Um dies zu vermeiden, ist es von Zeit zu Zeit erforderlich, die kritischen Datenstrukturen selbst zu sichern und die bis dahin angesammelten Nachrichten aus dem Eingangskanal des Managers zu entfernen. Beim Wiederanlauf muß der Manager dann die zuletzt gesicherten Daten einlesen und anschließend eventuell im Eingangskanal befindliche Nachrichten verarbeiten.

Anmerkung: Die Korrektheit von Variante 4 basiert auf der Annahme, daß die im Eingangskanal gespeicherten Nachrichten beim Wiederanlauf des Interaktionsmanagers in derselben Weise verarbeitet werden wie zuvor im Normalbetrieb. Für Exec- oder Undo-Bestätigungen, die im Normalbetrieb *zu spät* eingetroffen sind, trifft diese Annahme jedoch nicht zu, da sie sich beim Wiederanlauf bereits im Eingangskanal befinden und daher *nicht* als verspätet erkannt werden.

Da die Funktion `receive(m)` bei Zeitüberschreitung einfach die als Argument übergebene Nachricht `m` zurückliefert (vgl. § 5.2.9.2), ist es sinnvoll, diese zusätzlich in den persistenten Eingangskanal des Managers einzufügen. Beim Wiederanlauf findet `receive(m)` diese Nachricht dann *vor* einer eventuell verspätet eingetroffenen Exec- oder Undo-Nachricht, liefert sie zurück und verhält sich daher genau so wie zuvor im Normalbetrieb. Mit dieser einfachen Modifikation der Hilfsfunktion `receive()` kann die in § 5.2.9 beschriebene Implementierung somit unverändert übernommen werden.

5.3 Synchronisation paralleler Programme

5.3.1 Vorüberlegungen

Pfad- und Synchronisierungsausdrücke (vgl. § 6.2.1 und § 6.2.2) gehören zu den wenigen verwandten Ansätzen, die primär als Synchronisationsmechanismus für parallele oder verteilte Programmiersprachen entwickelt wurden. Verglichen mit primitiveren Mechanismen zur Synchronisation nebenläufiger Prozesse, wie z. B. Semaphoren [Dijkstra68ab] oder auch kritischen Abschnitten [BrinchHansen72], erlauben solche ausdrucksbasierten Formalismen eine wesentlich übersichtlichere und anwendungsorientiertere Formulierung von Synchronisationsbedingungen [Campbell74, Govindarajan91, Guo95]. Bei einer geeigneten Integration des Formalismus mit den übrigen Teilen einer Programmiersprache ist es außerdem möglich, Synchronisationsbedingungen vom eigentlichen Anwendungscode zu *trennen* bzw. sie vor diesem (im Sinne von information hiding) vollständig zu *verbergen* [Campbell74, Campbell79].

Ebenso wie Pfad- und Synchronisierungsausdrücke, lassen sich auch Interaktionsausdrücke prinzipiell in nahezu beliebige parallele bzw. entsprechend erweiterte sequentielle Programmiersprachen integrieren. Beispielsweise könnte man die momentan äußerst populäre Sprache *Java* [Flanagan98], die die Formulierung nebenläufiger Programme mit Hilfe von *Threads* unterstützt, problemlos dahingehend erweitern, daß Methoden oder Anweisungsblöcke nicht nur *synchronized* (d. h. als kritischer Abschnitt) ausgeführt werden können, sondern mit Hilfe von Interaktionsausdrücken wesentlich differenziertere Synchronisationsbedingungen für Methodenaufrufe formuliert werden können. Aufgrund der Implementierung von Interaktionsausdrücken in `CH`, bietet sich eine Integration mit dieser Sprache allerdings besonders an.

5.3.2 Integration von Interaktionsausdrücken in `CH`

Anhand des Beispiels der *speisenden Philosophen* (engl. dining philosophers) [Dijkstra71] sollen im folgenden die zusätzlichen Sprachkonstrukte einer entsprechend erweiterten Programmiersprache `CH` sowie die wesentlichen Implementierungskonzepte kurz erläutert werden.

Die speisenden Philosophen verbringen ihr Leben bekanntlich mit Nachdenken und Essen. Um essen zu können, betritt ein Philosoph den Speisesaal und setzt sich an seinen Platz an einem runden Tisch, in dessen Mitte sich eine Schüssel mit Spaghetti befindet (die niemals leer wird). Außerdem liegt für jeden Philosophen eine Gabel zu seiner linken bereit. Bedauerlicherweise sind die Spaghetti in der Schüssel aber so verschlungen, daß man sie nur mit *zwei* Gabeln essen kann. Daher benutzt jeder Philosoph zum Essen nicht nur seine eigene, sondern zusätzlich die Gabel seines rechten Nachbarn. Dies hat zur Folge, daß ein Philosoph nur essen kann, wenn die Gabeln zu seiner linken *und* zu seiner rechten gerade verfügbar sind, d. h. wenn seine *beiden* Nachbarn gerade *nicht* essen.

Nach dem Essen legt der Philosoph beide Gabeln zurück auf den Tisch und verläßt den Raum, um erneut nachzudenken.

5.3.2.1 Aktivitäten

Die Tätigkeiten der Philosophen, wie z. B. Nachdenken, Essen usw., können durch Prozeduren `think()`, `eat()` etc. modelliert werden, die als Parameter jeweils natürliche Zahlen zur Identifikation von Philosophen bzw. Gabeln erhalten (vgl. Abb. 5.24). Das vorangestellte Schlüsselwort `sync` − das eine ähnliche, aber nicht identische Bedeutung wie das Schlüsselwort `synchronized` der Sprache Java besitzt − signalisiert dem `CH`-Präcompiler, daß diese Funktionen in nachfolgenden *Synchronisationsbedingungen* verwendet werden können und deshalb anders als gewöhnliche Funktionen behandelt werden müssen.[11]

[11] Das Schlüsselwort `proc` steht für *procedure* und kennzeichnet eine Funktion ohne Rückgabewert (vgl. § A.2.3).

```
// Philosoph p denkt nach.
sync proc think(int p) {
    print "think(", p, ") ...";
    sleep(random());
    print newline;
}
// Philosoph p betritt den Speisesaal und setzt sich an den Tisch.
sync proc enter(int p) {
    print "enter(", p, ")", newline;
}
// Philosoph p nimmt Gabel f vom Tisch.
sync proc get(int p, int f) {
    print "get(", p, ", ", f, ")", newline;
}
// Philosoph p legt Gabel f zurück auf den Tisch.
sync proc put(int p, int f) {
    print "put(", p, ", ", f, ")", newline;
}
// Philosoph p ißt mit den Gabeln l und r.
sync proc eat(int p, int l, int r) {
    print "eat(", p, ", ", l, ", ", r, ") ...";
    sleep(random());
    print newline;
}
// Philosoph p verläßt den Raum.
sync proc leave(int p) {
    print "leave(", p, ")", newline;
}
```

Abbildung 5.24: Aktivitäten der Philosophen als zu synchronisierende Prozeduren

Ähnlich wie bei der Implementierung von Pfad- und Synchronisierungsausdrücken [Campbell74, Govindarajan90], werden derartige Funktionen im Prinzip mit einem *Prolog* und einem *Epilog* umgeben, die später für die korrekte Synchronisierung sorgen werden. Beispielsweise wird die vom Programmierer formulierte Funktion `get()` geeignet umbenannt (beispielsweise in `get_org()`) und durch eine neue Funktion `get()` mit derselben Signatur ersetzt, die vor und nach dem Aufruf der ursprünglichen Funktion `get_org()` jeweils die Hilfsfunktion `wait()` aufruft (vgl. Abb. 5.25). Diese erhält als Parameter zum einen den *Namen* der zu synchronisierenden Funktion als Zeichenkette (im Beispiel also `"get"`) und zum anderen die *Werte* ihrer aktuellen *Aufrufparameter*, die mit Hilfe einer geeignet zu definierenden (und i. d. R. statisch überladenen) Funktion `val()` bestimmt werden. (Für numerische Parameter liefert `val()` z. B. eine textuelle Repräsentation der jeweiligen Zahl.) Schließlich wird durch Übergabe der Zahl `0` oder `1` angezeigt, ob `wait()` am Anfang oder am Ende der zu synchronisierenden Funktion aufgerufen wird.

Betrachtet man eine Funktion wie z. B. `get()` als Aktivität A, so entsprechen die Aufrufe von `wait()` am Anfang bzw. Ende der Funktion der Ausführung der Aktionen A_0 bzw. A_1, die das Starten bzw. Beenden der Aktivität A markieren (vgl. § 2.6.2). Da die Ausführung dieser Aktionen von einem Interaktionsmanager verfolgt und kontrolliert werden muß (vgl. § 5.3.2.2), implementiert `wait()` das *einphasige Koordinationsprotokoll*, das für diese Anwendung ausreichend ist (vgl. § 5.2.2.2). Konkret fragt `wait()` mit Hilfe einer Wait-Nachricht beim Interaktionsmanager an, ob

```
// Philosoph p nimmt Gabel f vom Tisch.
proc get_org(int p, int f) {
    print "get(", p, ", ", f, ")", newline;
}

// Vom Präcompiler erzeugte Funktion.
proc get(int p, int f) {
    wait("get", val(p), val(f), 0);     // Prolog.
    get_org(p, f);                      // Originalfunktion.
    wait("get", val(p), val(f), 1);     // Epilog.
}
```

Abbildung 5.25: Prolog- und Epilogfunktionen

bzw. wann die entsprechende Aktion ausgeführt werden darf, und wartet anschließend auf eine Accept- (oder Unknown-) Antwort des Managers. Wenn diese eintrifft, hat der Manager einen entsprechenden Zustandsübergang durchgeführt, und `wait()` kehrt zu seinem Aufrufer zurück.

5.3.2.2 Synchronisationsbedingungen

Mit Hilfe des Schlüsselworts `expr` können nun *Synchronisationsbedingungen* für die zuvor formulierten Funktionen in Form von (linearisierten) Interaktionsausdrücken spezifiziert werden:[12]

```
expr +[f] * |[p] (get(p, f) - put(p, f));
expr +{4} * |[p] (enter(p) - leave(p));
```

Abbildung 5.26 zeigt die entsprechenden Interaktionsgraphen, die die folgenden Bedingungen spezifizieren:

1. Jede Gabel `f` darf immer nur von einem Philosophen `p` gleichzeitig verwendet werden, d. h. jeder Ausführung von `get(p, f)` (mit einer beliebigen Belegung des Parameters `p`) muß ein entsprechendes `put(p, f)` (mit demselben Wert für `p`) folgen, bevor das nächste `get(p, f)` (wieder mit einer beliebigen Belegung von `p`) ausgeführt werden darf (erster Ausdruck bzw. Graph).

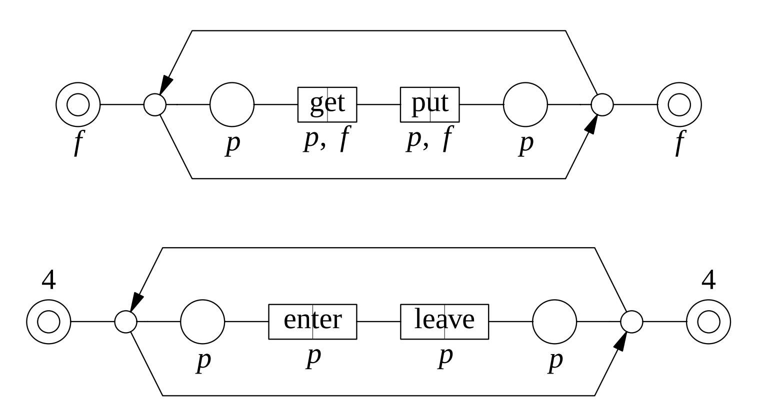


Abbildung 5.26: Synchronisationsbedingungen

[12] Da die zu synchronisierenden Funktionen zeitlich ausgedehnte Aktivitäten A darstellen, werden sie in einem Ausdruck implizit durch eine Folge zweier Aktionen A_0 und A_1 ersetzt (vgl. § 5.3.2.1).

2. Zu jedem Zeitpunkt dürfen sich maximal vier der fünf Philosophen gleichzeitig im Speisesaal befinden, d. h. die Sequenz `enter(p) − leave(p)` durchlaufen, weil andernfalls die Gefahr einer Verklemmung beim Zugriff auf die Gabeln besteht (zweiter Ausdruck bzw. Graph).

Man beachte in diesem Zusammenhang, daß die konkrete Anzahl von Philosophen bzw. Gabeln lediglich in den zweiten Ausdruck eingeht, der quasi einen „Türsteher“ beschreibt, der maximal vier Philosophen gleichzeitig in den Speisesaal einläßt. Der erste Ausdruck hingegen, der die korrekte Verwendung der Gabeln sicherstellt, hängt weder von einer bestimmten Philosophenzahl noch von der konkreten Zuordnung von Gabeln zu Philosophen ab. Diese wird erst später bei der Formulierung von *Prozessen* (§ 5.3.2.3) festgelegt.[13]

Der `CH`-Präcompiler verwandelt die mit `expr` vereinbarten Interaktionsausdrücke in Zeichenketten-Konstanten, die während der Initialisierungsphase des Programms (d. h. innerhalb eines `begin`-Blocks; vgl. § A.2.2) mit Hilfe einer Funktion `expr()` an den Interaktionsmanager übermittelt werden:[14]

```
begin {
    expr("+[f] * |[p] (get(p, f) − put(p, f))");
    expr("+{4} * |[p] (enter(p) − leave(p))");
}
```

Der Interaktionsmanager, der implizit durch den ersten Aufruf der Funktion `expr()` gestartet wird, verknüpft alle an ihn übergebenen Ausdrücke mittels einer Kopplung, die anschließend, wie in § 5.2.2.1 beschrieben, verarbeitet wird. Als Nachrichtenkanäle können entweder Hauptspeicher-Datenstrukturen oder einfache Interprozeßkommunikations-Mechanismen (wie z. B. Pipes, Message queues oder Sockets [Rochkind88, Stevens92]) verwendet werden, je nachdem, ob der Interaktionsmanager und seine Klienten als Threads innerhalb eines Prozesses oder als eigenständige Prozesse ablaufen (vgl. § 5.3.2.3). Auf eine persistente Speicherung von Nachrichten kann aus Performance-Gründen verzichtet werden, da bei einem Systemausfall i. d. R. alle beteiligten Prozesse komplett neu gestartet werden müssen.

5.3.2.3 Prozesse

Nachdem auf diese Weise sowohl die „elementaren Prozeßschritte“ (in Form von Prozeduren oder Funktionen) als auch die zugehörigen Synchronisationsbedingungen (in Form von Interaktionsausdrücken) spezifiziert wurden, benötigt man noch ein Sprachmittel zur Erzeugung *nebenläufiger Prozesse*. Hierfür kann das Schlüsselwort `proc`, das in `CH` eine Funktion ohne Rückgabewert kennzeichnet, durch das Wort `process` ersetzt werden, das besagt, daß jeder Aufruf der so definierten Funktion in einem *eigenen Prozeß* ausgeführt werden soll. Abbildung 5.27 zeigt eine derartige *Prozeßfunktion*, die den „Lebenszyklus“ eines Philosophen `p` beschreibt, der mit den Gabeln `l` und `r` ißt.

Auch diese Funktion wird vom `CH`-Präcompiler umbenannt und durch eine neue Funktion mit derselben Signatur ersetzt, die für die Erzeugung eines neuen Prozesses sorgt, wobei an dieser Stelle offenbleibt, ob es sich bei Prozessen um leichtgewichtige Threads oder um echte Betriebssystemprozesse auf einem oder mehreren Rechnern handelt. Zur Illustration zeigt Abb. 5.28, wie mit Hilfe des Unix-Systemaufrufs `fork()`[15] jeweils ein neuer Betriebssystemprozeß (auf demselben Rechner) erzeugt wird. Eine Prozeßfunktion kehrt also einerseits sofort zu ihrem Aufrufer zurück und erzeugt an-

[13] Um noch unabhängiger von der konkreten Anzahl `n` der Philosophen bzw. Gabeln zu sein, könnte man diese auch als Konstante vereinbaren (`const int n = 5`) und den zweiten Ausdruck dann als `expr +{n−1} ...` formulieren.

[14] Enthält ein Ausdruck, wie in der vorigen Fußnote erwähnt, symbolische Konstanten und/oder arithmetische Operatoren, so werden diese aus der Zeichenketten-Darstellung des Interaktionsausdrucks extrahiert und separat im Stil der C-Bibliotheksfunktion `printf()` übergeben: `expr("+{%d} ...", n−1);`

[15] Die Funktion `fork()` spaltet den aufrufenden Prozeß in zwei nahezu identische Prozesse auf, wobei im neu erzeugten („Kind-“) Prozeß der Wert 0 und im ursprünglichen („Vater-“) Prozeß ein positiver Wert (nämlich die Prozeßidentifikationsnummer des Kindprozesses) zurückgeliefert wird. Sollte `fork()` aufgrund mangelnder Betriebssystem-Ressourcen fehlschlagen, wird (im Vaterprozeß) der Wert `−1` zurückgeliefert (und kein Kindprozeß erzeugt) [Rochkind88].

```
// Lebenszyklus des Philosophen p, der mit den Gabeln l und r ißt.
process phil(int p, int l, int r) {
    loop {                  // Endlosschleife:
        think(p);               // Nachdenken.
        enter(p);               // Speisesaal betreten.
        get(p, l); get(p, r);   // Gabeln aufnehmen.
        eat(p, l, r);           // Essen.
        put(p, l); put(p, r);   // Gabeln weglegen.
        leave(p);               // Speisesaal verlassen.
    }
}
```

Abbildung 5.27: Lebenszyklus eines Philosophen als Prozeßfunktion

```
// Lebenszyklus des Philosophen p, der mit den Gabeln l und r ißt.
proc phil_org(int p, int l, int r) {
    loop {
        .....
    }
}

// Vom Präcompiler erzeugte Funktion.
proc phil(int p, int l, int r) {
    switch (fork()) {
    case -1:            // fork() schlug fehl:
        .....               // Geeignete Fehlerbehandlung.
        return;
    case 0:             // Neuer Prozeß:
        phil_org(p, l, r); // Prozeßfunktion ausführen
        exit(0);            // und Prozeß beenden.
    default:            // Ursprünglicher Prozeß:
        return;             // Zum Aufrufer zurückkehren.
    }
}
```

Abbildung 5.28: Transformation einer Prozeßfunktion

dererseits einen neuen Prozeß, der den Rumpf der Prozeßfunktion ausführt und anschließend wieder beendet wird.

Im Hauptprogramm (vgl. Abb. 5.29) werden durch fünf aufeinanderfolgende Aufrufe der Prozeßfunktion `phil()` fünf nebenläufige Philosophen-Prozesse erzeugt, bevor der Hauptprozeß selbst zu Ende geht.[16]

5.3.2.4 Beispielablauf

Abbildung 5.30 zeigt einen möglichen Ablauf der Prozesse `phil(1, 1, 2)` und `phil(2, 2, 3)`, wenn man vereinfachend annimmt, daß die anderen drei Prozesse erst später gestartet werden oder

[16] Auch dieser Programmteil ließe sich unabhängig von der konkreten Philosophenzahl `n` wie folgt formulieren (der Operator `%` bezeichnet die Modulo-Operation): `for(i = 1; i <= n; i++) phil(i, i, i%n + 1);`

```
begin {
    phil(1, 1, 2);      // Philosoph 1 ißt mit den Gabeln 1 und 2.
    phil(2, 2, 3);      // Philosoph 2 ißt mit den Gabeln 2 und 3.
    phil(3, 3, 4);      // Philosoph 3 ißt mit den Gabeln 3 und 4.
    phil(4, 4, 5);      // Philosoph 4 ißt mit den Gabeln 4 und 5.
    phil(5, 5, 1);      // Philosoph 5 ißt mit den Gabeln 5 und 1.
}
```

Abbildung 5.29: Hauptprogramm

momentan vom Betriebssytem suspendiert sind. Die Aufrufe der Hilfsfunktion `wait()` kommunizieren jeweils mit dem Interaktionsmanager, der durch die vertikale Linie zwischen den beiden Prozessen symbolisiert wird.

Beide Prozesse durchlaufen zunächst die Funktionen `think()` (die dem Manager unbekannt ist, da sie in keinem der beiden Interaktionsausdrücke aus § 5.3.2.2 vorkommt) und `enter()` sowie `get(1, 1)` bzw. `get(2, 2)`. Die zugehörigen Wait-Anfragen führen zu entsprechenden Zustandsübergängen im Interaktionsmanager und werden jeweils sofort mit Accept beantwortet, da bis zu diesem Zeitpunkt keinerlei Konflikte zwischen den Prozessen auftreten. Dasselbe gilt auch für alle weiteren Funktionsaufrufe des rechten Prozesses.

Die Anfrage der Funktion `wait("get", val(1), val(2), 0)` des linken Prozesses wird jedoch zunächst in die Warteliste des Managers eingetragen, da sich die Gabel 2 durch den vorangegangenen Aufruf der Funktion `get(2, 2)` des rechten Prozesses momentan im Besitz des Philosophen 2 befindet und der obere Graph in Abb. 5.26 daher die Ausführung von `get(1, 2)` verbietet. Erst nach Ausführung der Funktion `wait("put", val(2), val(2), 1)` kehrt dieser Graph in seinen Ausgangszustand zurück, in dem er die Ausführung von `get(1, 2)` wieder erlaubt. Daher erhält der linke Prozeß nun eine entsprechende Accept-Antwort, in deren Folge er die nachfolgenden Funktionen „ungestört“ durchlaufen kann.

5.3.3 Anmerkungen

Mit den soeben vorgestellten Spracherweiterungen stellt die Sprache CH sicherlich noch keine vollständige parallele Programmiersprache dar, da mit Hilfe von Interaktionsausdrücken lediglich *einer* von (mindestens) zwei wesentlichen Aspekten paralleler Programme, nämlich die *Synchronisation* nebenläufiger Prozesse, abgedeckt wird. Für den anderen wichtigen Aspekt, die *Kommunikation* von Prozessen, müßten ebenfalls geeignete Mechanismen (wie z. B. shared variables, message queues, remote procedure calls o. ä.) angeboten werden, damit die Sprache für reale Programmierprojekte eingesetzt werden kann. Außerdem wären Ausdrucksmittel wie `parbegin` und `parend` [Dijkstra68a] sinnvoll, mit denen Parallelität nicht nur für komplette Prozeduren, sondern auch für einzelne Anweisungen formuliert werden kann.

Allerdings war die Entwicklung einer „YAPPL“ (yet another parallel programming language) nie ein Ziel dieser Arbeit. Vielmehr sollte die in diesem Abschnitt vorgestellte „PMPPL“ (poor man’s parallel programming language) lediglich verdeutlichen, wie Interaktionsausdrücke (ähnlich wie Pfad- und Synchronisierungsausdrücke) *prinzipiell* als „High-level-Synchronisationsmechanismus“ in parallelen Programmiersprachen eingesetzt werden können. Außerdem wird das in § 5.3.2 verwendete „Vorgehensmodell“ zur Entwicklung paralleler Programme − Definition von Aktivitäten, Definition von Synchronisationsbedingungen und schließlich Definition von Prozessen − im nachfolgenden Abschnitt 5.4 wiederaufgegriffen, um Systeme nebenläufiger Workflows zu entwickeln.

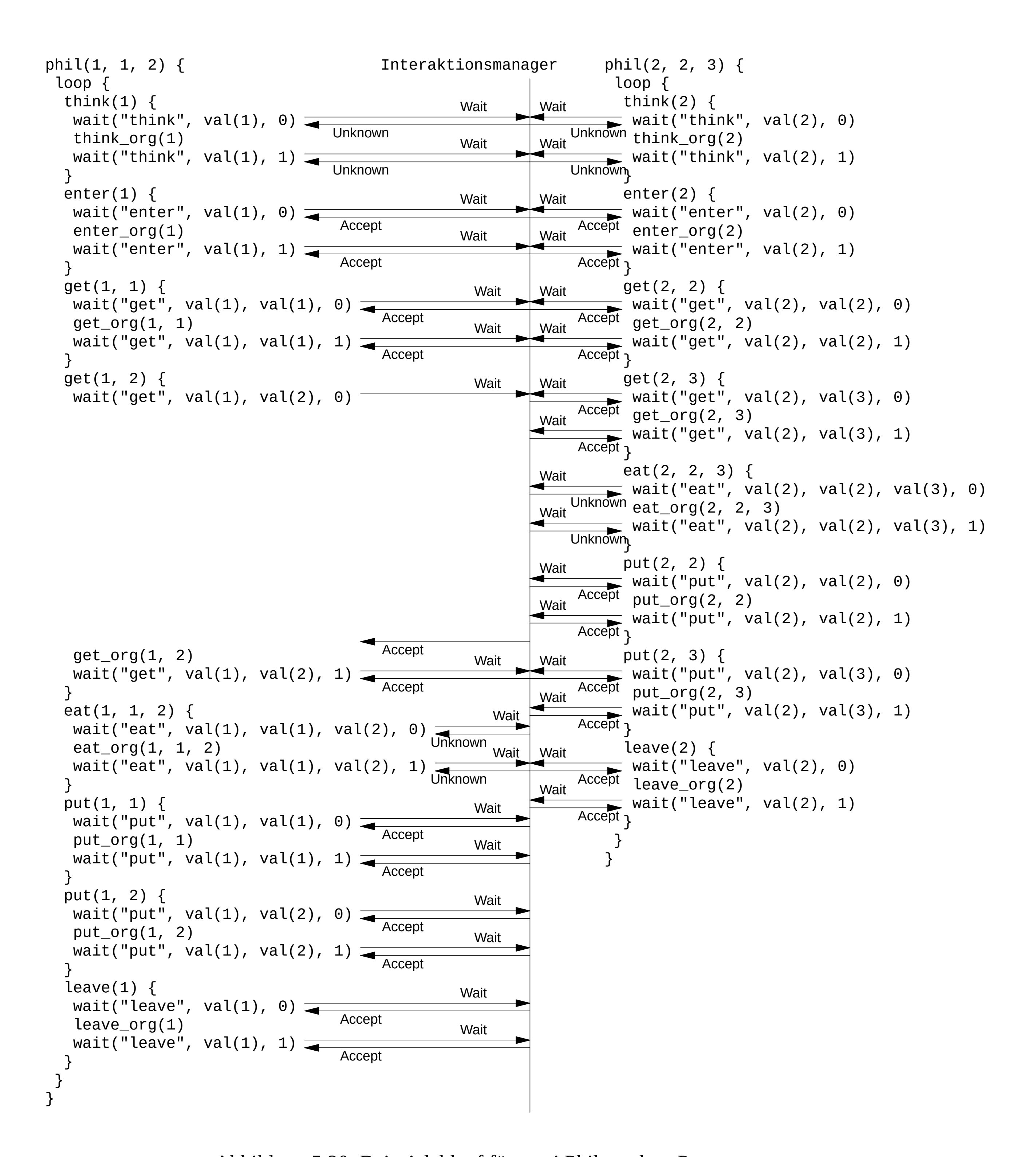


Abbildung 5.30: Beispielablauf für zwei Philosophen-Prozesse

5.4 Definition von Workflow-Geflechten

5.4.1 Vorüberlegungen

Eine Menge von Workflows, die potentiell überlappend ausgeführt werden, zusammen mit einer Menge von Inter-Workflow-Abhängigkeiten, die hierbei zu berücksichtigen sind, wird im weiteren als *Workflow-Geflecht* bezeichnet. Wie die folgenden Überlegungen zeigen, weist ein solches Workflow-Geflecht einige Parallelen zu einem nebenläufigen Programm auf.

Beispielsweise kann man die elementaren Schritte oder Aktivitäten eines Workflows durchaus mit Prozeduren oder Funktionen einer imperativen Programmiersprache vergleichen: Beide werden durch einen eindeutigen Namen identifiziert (sofern man von der Möglichkeit des statischen Überladens von Funktionsnamen absieht), besitzen ggf. eine Menge von Aufrufparametern, und bei ihrem Aufruf wird eine bestimmte, klar umrissene Tätigkeit ausgeführt.

Ein einzelner Workflow entspricht dann in verschiedener Hinsicht einem (mehr oder weniger) sequentiellen Programm oder Prozeß: Zur Definition eines Workflows werden i. d. R. dieselben oder ähnliche Kontrollkonstrukte wie in imperativen Programmiersprachen verwendet, nämlich Sequenz, bedingte Verzweigung und ggf. verschiedene Arten von Schleifen. Die Strukturierung eines größeren Workflows in überschaubare Blöcke oder Teilworkflows entspricht einer schrittweisen Verfeinerung eines imperativen Programms unter Verwendung von Unterprogrammen, d. h. Prozeduren oder Funktionen. Und schließlich wird der Begriff *Prozeß* sowohl für die Ausführung eines sequentiellen Programms als auch für die Ausführung eines Workflows verwendet. (Allerdings ist der Begriff im Kontext von Workflow-Management nicht ganz einheitlich definiert.)

Somit entspricht ein Workflow-Geflecht, d. h. eine Menge parallel auszuführender und ggf. zu synchronisierender Workflows, einem parallelen oder verteilten Programm, d. h. einer Menge parallel laufender Threads oder Betriebssystem-Prozesse mit zugehörigen Synchronisationsbedingungen.

5.4.2 Schritt 1: Erstellung eines Aktivitätenkatalogs

Aufgrund dieser Analogien ist es naheliegend, bei der Definition eines Workflow-Geflechts ähnlich vorzugehen wie in § 5.3.2 bei der Erstellung eines parallelen Programms, d. h. mit der Definition der elementaren Prozeß- oder Workflowschritte zu beginnen. Diese Vorgehensweise hat den Vorteil, daß man anschließend − bei der Definition von Workflows und Inter-Workflow-Abhängigkeiten − ein einheitliches und kontrolliertes „Vokabular“ von Aktivitätenbezeichnern zur Verfügung hat, auf das man sich beziehen kann.

Wie bereits erwähnt, besitzt ein Workflowschritt oder eine Aktivität − ebenso wie eine Prozedur − einen eindeutigen *Namen* und ggf. eine Menge von *Parametern*. Anstelle eines Prozedurrumpfs, der direkt den ausführbaren Code der jeweiligen „Tätigkeit“ enthält, muß einem Workflowschritt jedoch ein *Schrittprogramm* zugeordnet werden, d. h. ein beliebiges, meist interaktives Programm, mit dessen Hilfe der *Bearbeiter* des Schritts die auszuführende Tätigkeit verrichten kann (vgl. § 1.1.1). Darüber hinaus ist es sinnvoll, für jede Aktivität eine verbale *Beschreibung* dieser Tätigkeit zu formulieren, ebenso wie man Prozeduren üblicherweise mit einem erläuternden Kommentar versieht. Faßt man diese (und ggf. weitere relevante) Daten für alle Workflowschritte in einem zentralen Verzeichnis zusammen, so erhält man einen *Aktivitätenkatalog*.

Tabelle 5.31 zeigt einen solchen (stark vereinfachten) Katalog für den Anwendungsbereich „Medizinische Untersuchungen“. Die genannten Bezeichnungen der Schrittprogramme, wie z. B. elektronisches Kardex[17], elektronischer Kalender, Pflegedokumentation usw., sollen lediglich andeuten, um welche Art von Programmen es sich hierbei handeln könnte. Die Beschreibung der Tätigkeiten ist aus Platzgründen äußerst kurz gefaßt.

[17] Im Kardex (Krankenblatt) eines Patienten werden neben Blutdruck- und Fieberkurven auch sämtliche ärztlichen Anordnungen festgehalten.

Aktivität	*Parameter*	*Bearbeiter*	*Programm*	*Beschreibung*
Untersuchung anordnen	p, u	Stationsarzt	elektron. Kardex	Untersuchung u für Patient p anordnen.
Termin vereinbaren	p, u	Pflegekraft	elektron. Kalender	Termin für Untersuchung u des Patienten p vereinbaren.
Patient vorbereiten	p, u	Pflegekraft	Pflege-doku.	Patient p für Untersuchung u vorbereiten.
Patient aufklären	p, u	Stationsarzt	elektron. Akte	Patient p für Untersuchung u aufklären.
Patient abrufen	p, u	Med.-techn. Assistentin	Abruf-system	Patient p zur Untersuchung u abrufen.
Untersuchung durchführen	p, u	Untersuchender Arzt	Unters.-doku.	Untersuchung u für Patient p durchführen.
(Kurz-/Lang-) Befund erstellen	p, u	Untersuchender Arzt	Text-verarb.	Befund über Untersuchung u des Patienten p erstellen.
(Kurz-/Lang-) Befund lesen	p, u	Stationsarzt	elektron. Akte	Befund über Untersuchung u des Patienten p lesen.

Tabelle 5.31: Aktivitätenkatalog

Anmerkung: Normalerweise sollte der Einführung eines Workflow-Management-Systems in einer Klinik oder einem anderen Unternehmen eine umfassende Analyse und ggf. Restrukturierung aller wichtigen Geschäftsprozesse vorausgehen, die die Erstellung eines solchen Katalogs i. d. R. miteinschließt.

5.4.3 Schritt 2: Spezifikation allgemeiner Integritätsbedingungen

Nach diesem ersten, grundlegenden Schritt könnte man prinzipiell mit der Definition einzelner Workflows (in einer geeigneten Workflow-Beschreibungssprache) fortfahren und abschließend die zugehörigen Inter-Workflow-Abhängigkeiten (mit Hilfe von Interaktionsausdrücken oder -graphen) spezifizieren.

Bei der Formulierung von Inter-Workflow-Abhängigkeiten stellt man jedoch fest, daß sich diese prinzipiell in zwei Klassen einteilen lassen: Zum einen gibt es *allgemeine Integritätsbedingungen*, wie z. B. die in § 2.7.1.2 entwickelte Integritätsbedingung für Patienten, die einfach in der „Natur" der einzelnen Aktivitäten begründet liegen und nichts mit ihrer Verwendung in konkreten Workflow-Definitionen zu tun haben. Derartige Bedingungen könnte man in gewisser Weise sogar als Teil des Aktivitätenkatalogs auffassen, ebenso wie Integritätsbedingungen in Datenbanken einen Teil der Schemadefinition darstellen [Reuter87, Date98]. Auf der anderen Seite gibt es *spezielle Integritätsbedingungen*, wie z. B. die in § 2.7.4.3 genannte Reihenfolgebeziehung zwischen Sonographie und Endoskopie, die sich auf die Verwendung von Aktivitäten in ganz bestimmten Workflow-Definitionen beziehen.

Da allgemeine Integritätsbedingungen unabhängig von konkreten Workflow-Definitionen formuliert werden können, ist es sinnvoll, sie unmittelbar nach oder eventuell sogar zusammen mit der Erstellung des Aktivitätenkatalogs zu spezifizieren und erst im Anschluß daran mit der Definition konkreter Workflows fortzufahren. Diese Vorgehensweise besitzt außerdem den Vorteil, daß bestimmte Einschränkungen (z. B. daß ein Patient nicht gleichzeitig für eine Untersuchung vorbereitet und aufgeklärt werden kann) nicht in jeder Workflow-Definition erneut formuliert werden müssen, wenn sie zuvor *einmal* im Rahmen einer allgemeinen Integritätsbedingung spezifiziert wurden. Das heißt, daß derartige Bedingungen, die primär als *Inter*-Workflow-Abhängigkeiten gedacht sind, auch als zusätzliche *Intra*-Workflow-Abhängigkeiten verwendet werden können.

Die Abbildungen 5.32 bis 5.34 zeigen einige (bereits aus § 2.7 bekannte) allgemeine Integritätsbedingungen für die in Tab. 5.31 definierten Aktivitäten.

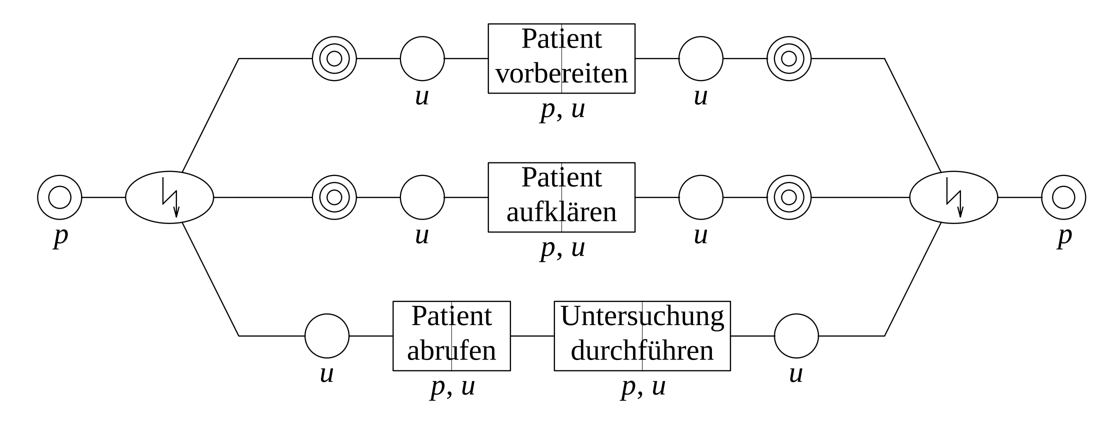


Abbildung 5.32: Integritätsbedingung für Patienten (vgl. Abb. 2.62, § 2.7.1.2)

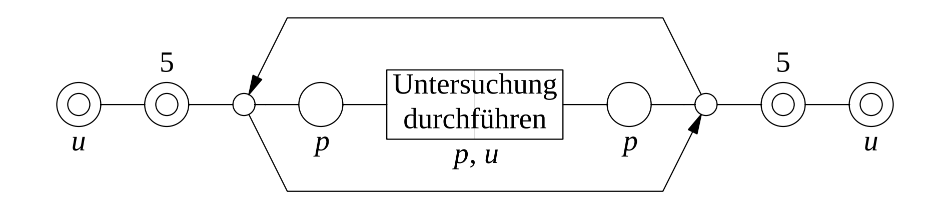


Abbildung 5.33: Allgemeine Kapazitätsbeschränkung für Untersuchungsstellen (vgl. Abb. 2.64, § 2.7.2.1)

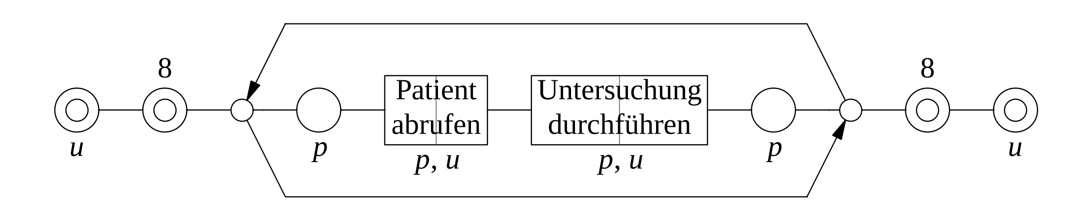


Abbildung 5.34: Begrenzung von Warteschlangen in Untersuchungsstellen (vgl. Abb. 2.68, § 2.7.3)

5.4.4 Schritt 3: Definition von Workflows

Im Anschluß an die Definition von Aktivitäten und allgemeinen Integritätsbedingungen können nun konkrete Workflows definiert werden. Die Abbildungen 5.35 und 5.36 zeigen exemplarisch die bereits in § 1.1.5 vorgestellten Workflows zur Durchführung einer sonographischen bzw. endoskopischen Untersuchung einschließlich aller hierfür erforderlichen Vor- und Nachbereitungen.

Da die allgemeine Integritätsbedingung für Patienten (Abb. 5.32) den wechselseitigen Ausschluß der Aktivitäten Patient vorbereiten und Patient aufklären sicherstellt, können diese Schritte im Endoskopie-Workflow einfach parallel angeordnet werden, obwohl sie, wie bereits erwähnt, nicht gleichzeitig ausgeführt werden dürfen. Ohne diese Bedingung müßte man in der Definition des Workflows eine bestimmte Reihenfolge für diese Schritte vorschreiben, da sich eine *beliebige Reihenfolge* von Schritten mit den üblichen imperativen Kontrollkonstrukten nicht formulieren läßt (vgl. auch § 1.3.3.2).

Da die Modellierung des Datenflusses von verschiedenen Workflow-Management-Systemen sehr unterschiedlich gehandhabt wird und Details an dieser Stelle nicht relevant sind, wird im folgenden nur vorausgesetzt, daß die im Aktivitätenkatalog für einen Schritt spezifizierten Parameter bei der

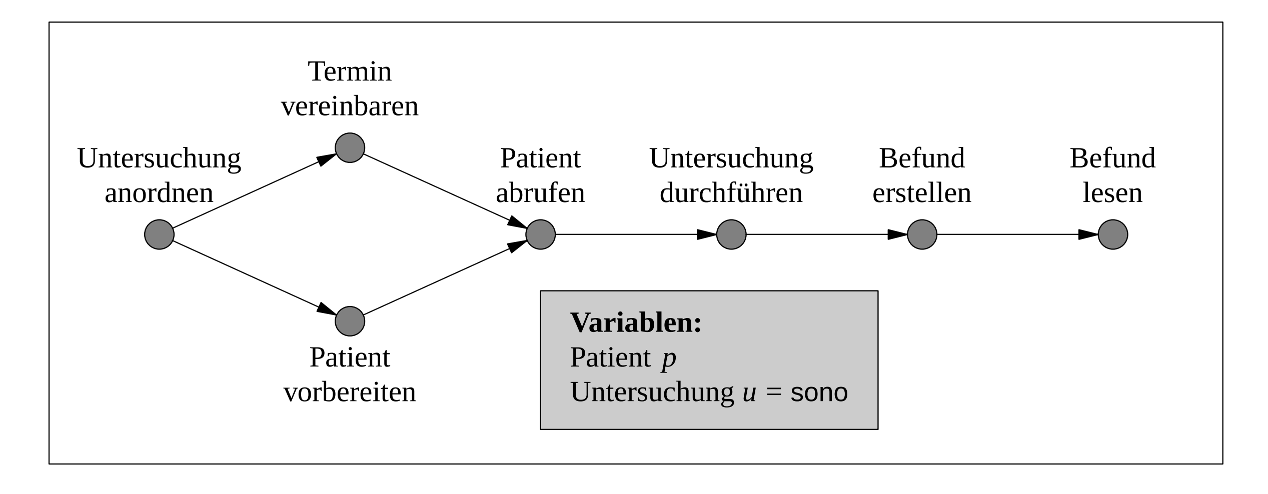


Abbildung 5.35: Sonographische Untersuchung

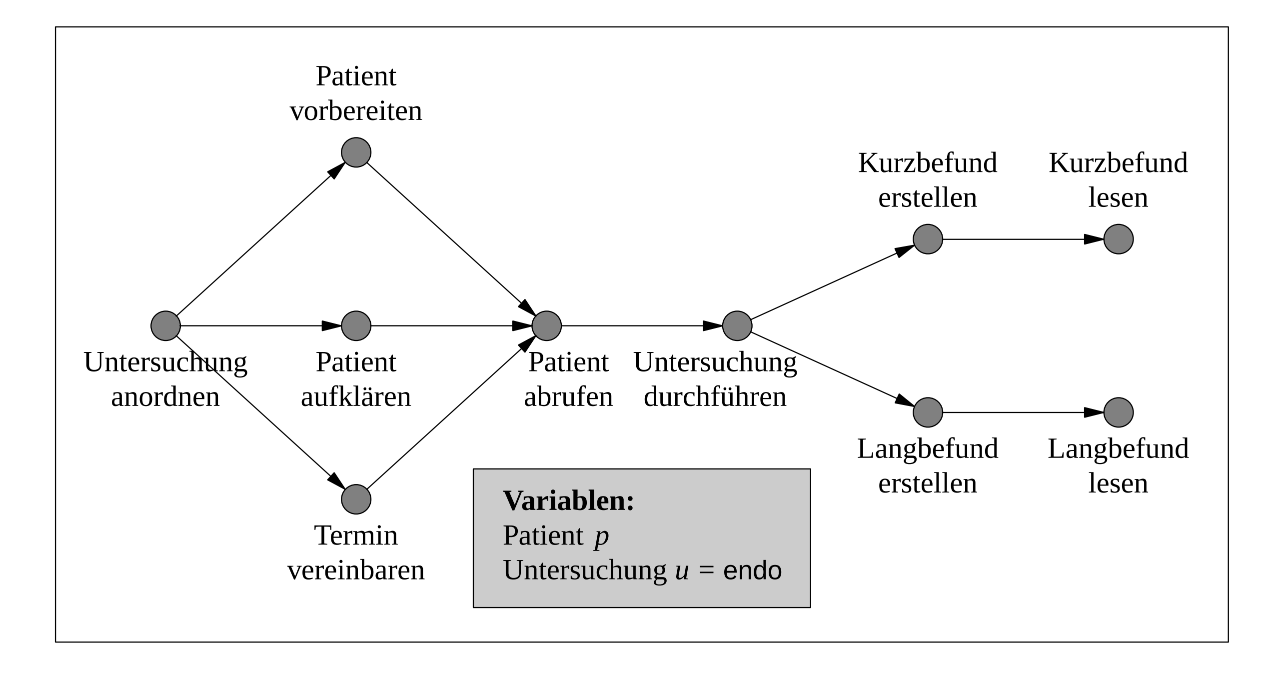


Abbildung 5.36: Endoskopische Untersuchung

Ausführung dieses Schritts über eine geeignete Schnittstelle zugreifbar sind. Als mentales Modell wird hierfür angenommen, daß die beiden Workflows jeweils „globale Variablen“ p und u besitzen, die beim Start des Workflows initialisiert und dann an jeden Workflowschritt als Parameter übergeben werden. Der Wert von u ist hierbei, je nach Typ des Workflows, eine der Konstanten sono oder endo, während der Parameter p in geeigneter Weise einen Patienten identifiziert, also z. B. eine Patienten-Identifikationsnummer enthält (vgl. auch § 2.7.1.2).

5.4.5 Schritt 4: Spezifikation spezieller Integritätsbedingungen

Wie bereits erwähnt, benötigt man neben den in Schritt 2 spezifizierten allgemeinen Integritätsbedingungen gelegentlich auch *spezielle Integritätsbedingungen*, die sich auf die Verwendung von Aktivitäten in ganz bestimmten Workflow-Definitionen beziehen. Derartige Bedingungen können natürlich

erst *nach* oder zusammen mit der Definition der entsprechenden Workflows formuliert werden und stellen daher den vierten und letzten Schritt bei der Definition eines Workflow-Geflechts dar.

Die Abbildungen 5.37 bis 5.39 zeigen die Definition der Abkürzung Mindestabstand (die eigentlich in einer globalen *Makrobibliothek* stehen sollte, siehe § 5.4.6), ihre Anwendung auf die Untersuchungsarten Endoskopie und Sonographie sowie die ebenfalls bereits bekannte spezielle Kapazitätsbeschränkung für die Sonographie.

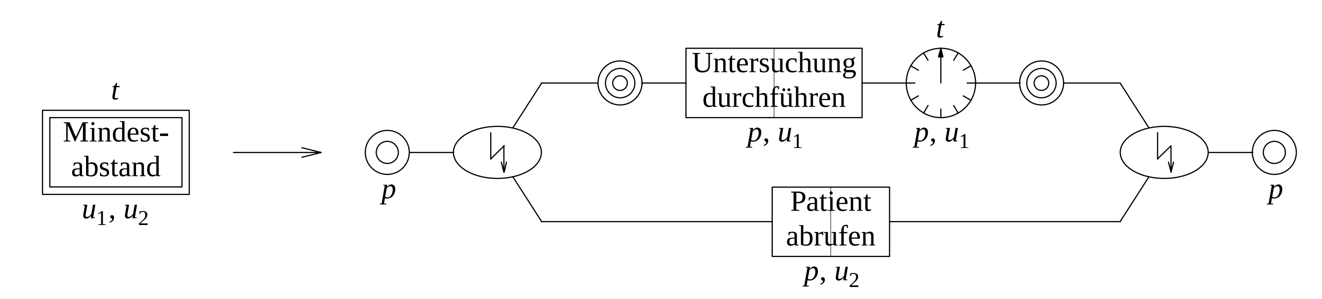


Abbildung 5.37: Definition der Abkürzung Mindestabstand (vgl. Abb. 2.71, § 2.7.4.4)

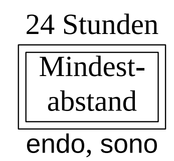


Abbildung 5.38: Mindestabstand zwischen Endoskopie und Sonographie (vgl. Abb. 2.72, § 2.7.4.4)

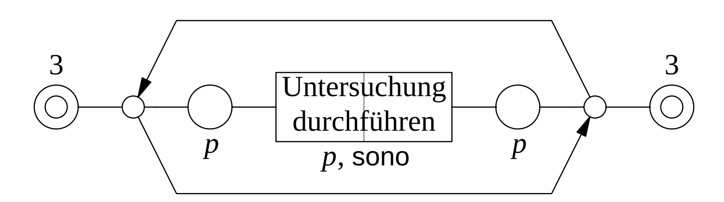


Abbildung 5.39: Spezielle Kapazitätsbeschränkung für die Sonographie (vgl. Abb. 2.65, § 2.7.2.2)

5.4.6 Zusammenfassung des Vorgehensmodells

Abbildung 5.40 zeigt die einzelnen Schritte zur Definition eines Workflow-Geflechts noch einmal im Zusammenhang. Außerdem wird angedeutet, daß sich die zur Erstellung von Interaktionsgraphen und Workflow-Definitionen verwendeten Editoren beide auf den zuvor erstellten Aktivitätenkatalog „abstützen“ und nur die dort definierten Namen zur Bezeichnung von Aktivitäten erlauben sollten. Für den in Anhang C beschriebenen Interaktionsgraph-Editor ist dieser Katalog-Anschluß tatsächlich implementiert, da Aktivitäten dort nur aus einer vorgegebenen Palette ausgewählt und nicht frei definiert werden dürfen (vgl. § C.1). Ob ein Standard-Workflow-Editor, d. h. die *Build-time-Komponente* eines Workflow-Management-Systems eine solche *A-priori-Definition* von Aktivitäten unterstützt, hängt stark vom jeweiligen System ab. Die meisten heutigen Systeme verleiten eher zu einer *En-passant-Definition* von Schritten während der Definition eines Workflows und unterstützen meist *keine Wiederverwendung* einmal definierter Schritte. In diesem Fall ist die in der Abbildung angedeutete Verbindung zwischen Aktivitätenkatalog und Workflow-Editor nur konzeptioneller Art und muß vom Modellierer durch ein entsprechend diszipliniertes Verhalten praktisch umgesetzt werden.

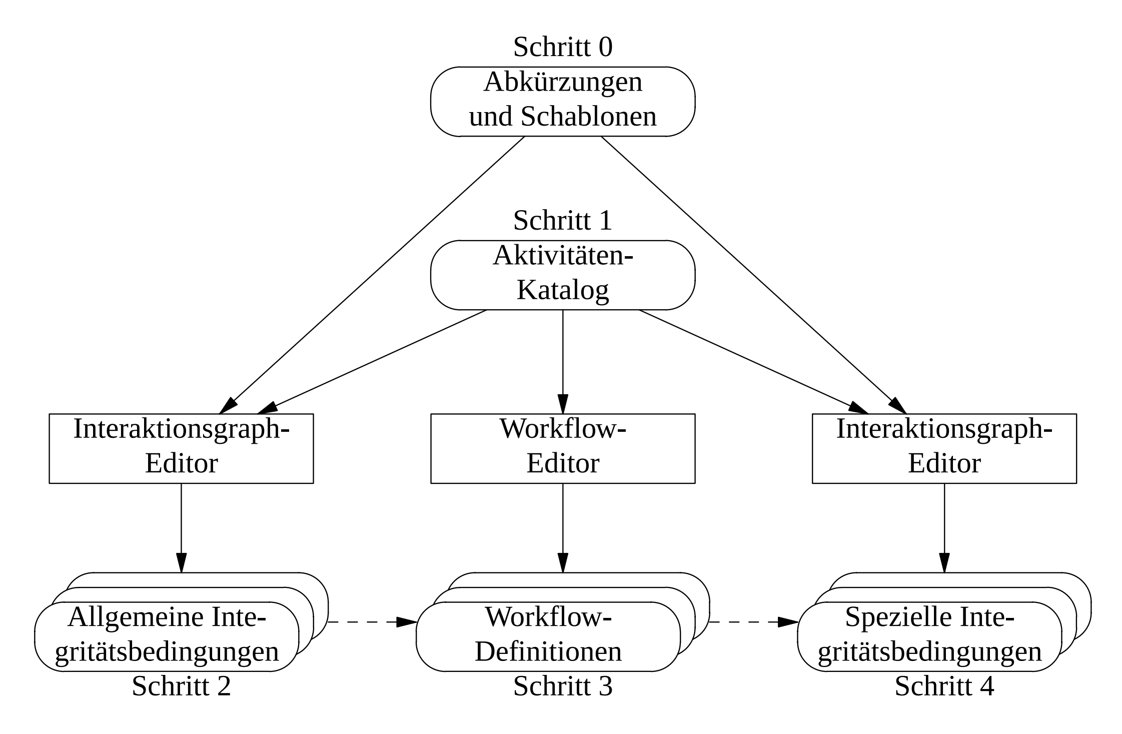


Abbildung 5.40: Vorgehensmodell zur Definition von Workflow-Geflechten

Darüber hinaus zeigt die Abbildung, daß in einem Schritt 0 (der ggf. auch parallel zu Schritt 1 ausgeführt werden kann) eine Bibliothek von Abkürzungen und Schablonen für Interaktionsgraphen erstellt werden sollte, die in den nachfolgenden Schritten 2 und 4 verwendet werden kann. Wie in § 2.8.2.2 und § 3.3.2.2 erläutert wurde, werden Makros wie z. B. Mindestabstand, deren vergleichsweise komplexe interne Struktur nicht unmittelbar ihre an sich einfache Bedeutung und Verwendung widerspiegelt, typischerweise von einem „Interaktionsgraph-Experten“ erstellt, und können dann auch von weniger geübten Anwendern, wie z. B. Workflow-Modellierern, benutzt werden.

5.4.7 Anmerkungen

Ebenso wie die bekannten Phasenmodelle des Software-Engineering [Schulz90, Denert91], sollte auch das hier beschriebene Vorgehensmodell zur Definition von Workflow-Geflechten nicht als starres Schema, sondern als sinnvolle Richtschnur verstanden werden. In der Praxis laufen die Schritte 1−4 sicherlich nicht immer streng sequentiell, sondern teilweise auch überlappend oder iterativ ab.

Möglicherweise ist man aber auch mit der unangenehmen Situation konfrontiert, daß bereits Workflow-Definitionen mit „inkompatiblen“ Aktivitäten-Bezeichnern existieren, was zumeist daher rührt, daß heutige Workflow-Systeme, wie bereits erwähnt, die hier propagierte A-priori-Definition von Arbeitsschritten in einem Katalog nicht oder nur unzureichend unterstützen. In einem solchen Fall muß man sich grundsätzlich mit denselben (und zum Teil sehr schwierigen) Problemen auseinandersetzen wie bei der *Schemaintegration* verteilter Datenbanken [Dadam96, Özsu91]. Neben der Existenz von *Synonymen* (d. h. unterschiedlichen Bezeichnungen für dieselbe Aktivität) und *Homonymen* (d. h. gleichen Bezeichnungen für verschiedene Aktivitäten) ist man u. U. auch mit inkompatiblen Parameterlisten, unterschiedlichen „Aktivitätengranulaten“ o. ä. konfrontiert.

In einem solchen Fall ist es sicherlich sinnvoll, *nachträglich* einen Aktivitätenkatalog zu erstellen, der neben dem *primären Namen* für jede Aktivität (der für zukünftige Definitionen verbindlich ist) auch eine Liste von *alternativen Bezeichnungen* enthält, mit deren Hilfe das Synonymproblem prinzipiell gelöst werden kann. Das Homonymproblem, das i. d. R. erheblich seltener auftritt, kann normalerweise dadurch gelöst werden, daß man neben dem Namen einer Aktivität auch das ihr zugeordnete *Schrittprogramm* zur Unterscheidung heranzieht, da unterschiedliche Aktivitäten in aller Regel auch

unterschiedliche Schrittprogramme besitzen. Zur Auflösung weitergehender Inkompatibilitäten lassen sich (ebenso wie für die schwierigeren Probleme der DB-Schemaintegration) meist keine fertigen „Rezepte" formulieren; vielmehr muß hier in jedem Einzelfall geprüft werden, ob und ggf. wie vorliegende Workflow-Definitionen geeignet angepaßt werden können, damit sie in das Gesamtschema integriert werden können.

5.5 Implementierung von Workflow-Geflechten

5.5.1 Überblick

Nach den vorangegangenen konzeptionellen Überlegungen zur Definition von Workflow-Geflechten, sollen im folgenden zwei konkrete Alternativen für ihre Implementierung vorgestellt werden.

Abschnitt 5.5.2 beschreibt, wie die Einhaltung Workflow-übergreifender Integritätsbedingungen − quasi auf Anwendungsebene − durch den Einsatz *adaptierter Arbeitslistenprogramme* gewährleistet werden kann, während § 5.5.3 die Verwendung *adaptierter Workflow-Ausführungseinheiten* erläutert. In beiden Abschnitten wird jeweils beschrieben, wie die Arbeitslisten von Benutzern aktualisiert werden und welche Protokollschritte beim Starten und Beenden von Aktivitäten durchlaufen werden müssen. Außerdem wird jeweils erläutert, wie sich die Prinzipien von einem einfachen System mit einer Workflow-Ausführungseinheit und einem Interaktionsmanager auf Systeme mit mehreren derartigen Komponenten verallgemeinern lassen.

Nach einer zusammenfassenden Diskussion der beiden Alternativen (§ 5.5.4), werden die Prinzipien in § 5.5.5 anhand eines praktischen Beispiels illustriert.

5.5.2 Adaption von Arbeitslistenprogrammen

5.5.2.1 Aktualisierung von Arbeitslisten

Wie in § 1.1.2 erläutert wurde, verwaltet ein Workflow-Management-System (genauer: die zentrale Ausführungseinheit; vgl. Abb. 1.3, § 1.1.6) für jeden Benutzer eine *Arbeitsliste*, in die alle Aktivitäten eingetragen werden, die von diesem Bearbeiter − aus Sicht des WfMSs − momentan ausgeführt werden dürfen. Mit Hilfe eines *Arbeitslistenprogramms* ist ein Benutzer einerseits in der Lage, seine Arbeitsliste einzusehen, und andererseits, darin angebotene Aktivitäten zu starten. Würde ein Arbeitslistenprogramm bestimmte Aktivitäten bei der Anzeige einer Arbeitsliste unterdrücken oder ihre Ausführung verweigern, so wäre der Benutzer nicht in der Lage, diese Aktivitäten zu starten.

Diese Tatsache kann unmittelbar dazu verwendet werden, Workflow-übergreifende Integritätsbedingungen zu implementieren, ohne in die Definition der zu synchronisierenden Workflows oder in die zentrale Ausführungseinheit des WfMSs eingreifen zu müssen. Hierzu wird die Funktionsweise eines Arbeitslistenprogramms wie folgt modifiziert (vgl. Abb. 5.41):

- Für jede Aktivität A, die aus Sicht der Workflow-Ausführungseinheit momentan zulässig ist (1), wird eine Subscribe-Nachricht für die Startaktion A_0 an den zuständigen Interaktionsmanager geschickt (2).[18]
- Erst wenn vom Manager eine Permit-Nachricht für die Aktion A_0 eintrifft (3), wird die Aktivität A tatsächlich in der Arbeitsliste des Benutzers angezeigt (4), weil sie erst dann wirklich ausgeführt werden darf. Wenn später eine zugehörige Forbid-Nachricht eintrifft (5), wird die Aktivität wieder aus der Arbeitsliste entfernt (6), weil sie dann (vorübergehend) nicht ausgeführt werden darf. (Alternativ könnte man unzulässige Aktivitäten auch in der Arbeitsliste belassen und sie lediglich als nicht ausführbar kennzeichnen.)
  Dieses Wechselspiel von Permit- und Forbid-Nachrichten kann sich beliebig oft wiederholen (3−6).

[18] Zur Vereinfachung der Darstellung wird zunächst davon ausgegangen, daß es nur einen Interaktionsmanager gibt. Alle Überlegungen lassen sich jedoch ohne weiteres auf ein System mit mehreren Managern verallgemeinern (vgl. § 5.5.2.4).

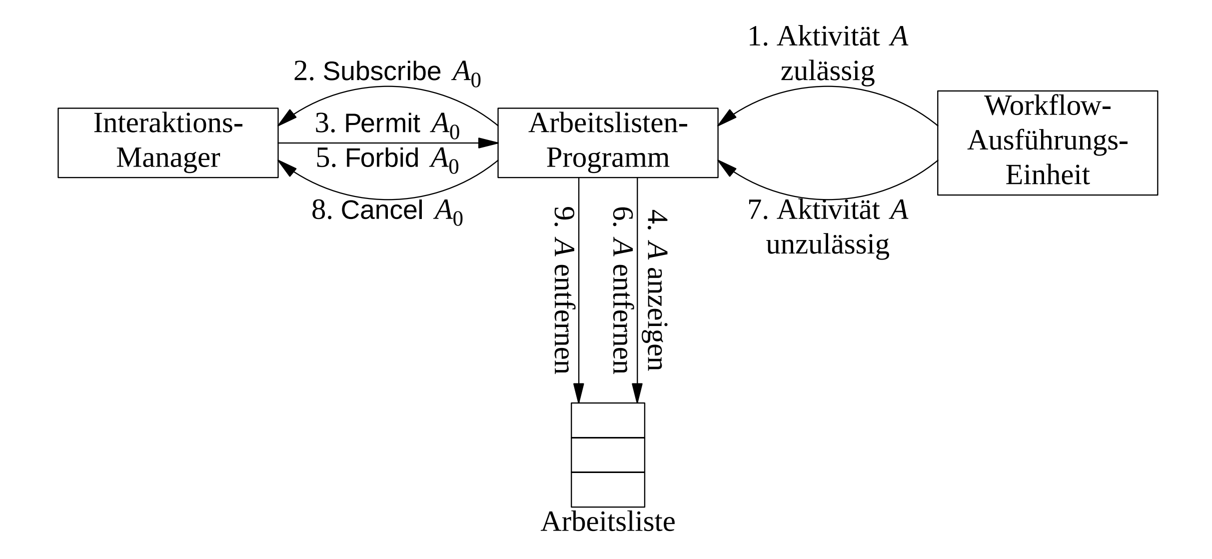


Abbildung 5.41: Aktualisierung einer Arbeitsliste

- Ist eine Aktivität A aus Sicht der Workflow-Ausführungseinheit nicht mehr zulässig (7) (typischerweise weil sie von einem anderen Bearbeiter gestartet wurde), so sendet das Arbeitslistenprogramm eine Cancel-Nachricht für die Aktion A_0 an den Interaktionsmanager (8), um die Permit- und Forbid-Nachrichten für diese Aktion abzubestellen. Außerdem wird die Aktivität aus der Arbeitsliste entfernt, sofern sie momentan darin enthalten ist (9).

Anstelle einer Permit-Nachricht könnte in Schritt 3 auch eine Unknown-Nachricht des Managers für die Aktion A_0 eintreffen, die ebenfalls dazu führt, daß die Aktivität A in der Arbeitsliste angezeigt wird (4) und so lange darin verbleibt, bis sie aus Sicht der Workflow-Ausführungseinheit nicht mehr zulässig ist (7−9).

5.5.2.2 Starten von Aktivitäten

Will ein Benutzer eine Aktivität A tatsächlich starten, so verfährt das Arbeitslistenprogramm gemäß des dreiphasigen Koordinationsprotokolls, das im Normalfall wie folgt abläuft (vgl. Abb. 5.42):

- Mit Hilfe einer Ask-Nachricht wird beim Interaktionsmanager angefragt, ob die Aktion A_0 tatsächlich ausgeführt werden darf (1).
- Bei Eintreffen der Accept-Antwort (2) wird die Workflow-Ausführungseinheit aufgefordert, die Aktivität A zu starten (3).
- Wird dies von der Ausführungseinheit bestätigt (4), so wird eine Exec-Bestätigung für die Aktion A_0 an den Manager gesandt (5).
- Bei Eintreffen der Commit-Nachricht (6) wird das zur Aktivität A gehörende Schrittprogramm tatsächlich gestartet (7).

Die folgenden Abweichungen vom Normalfall müssen jedoch berücksichtigt werden:

- Trifft in Schritt 2 anstelle der Accept- eine Unknown-Nachricht ein, so entfallen die Schritte 5 und 6.
- Sollte in Schritt 2 weder eine Accept- noch eine Unknown-, sondern eine Reject-Nachricht eintreffen, so darf die Aktivität nicht gestartet werden (typischerweise weil sie gerade von einem anderen Bearbeiter gestartet wurde). In diesem Fall entfallen die Schritte 3 bis 7. Quasi gleichzeitig mit der Reject-Nachricht erhält das Arbeitslistenprogramm vom Interaktionsmanager auch eine Forbid-

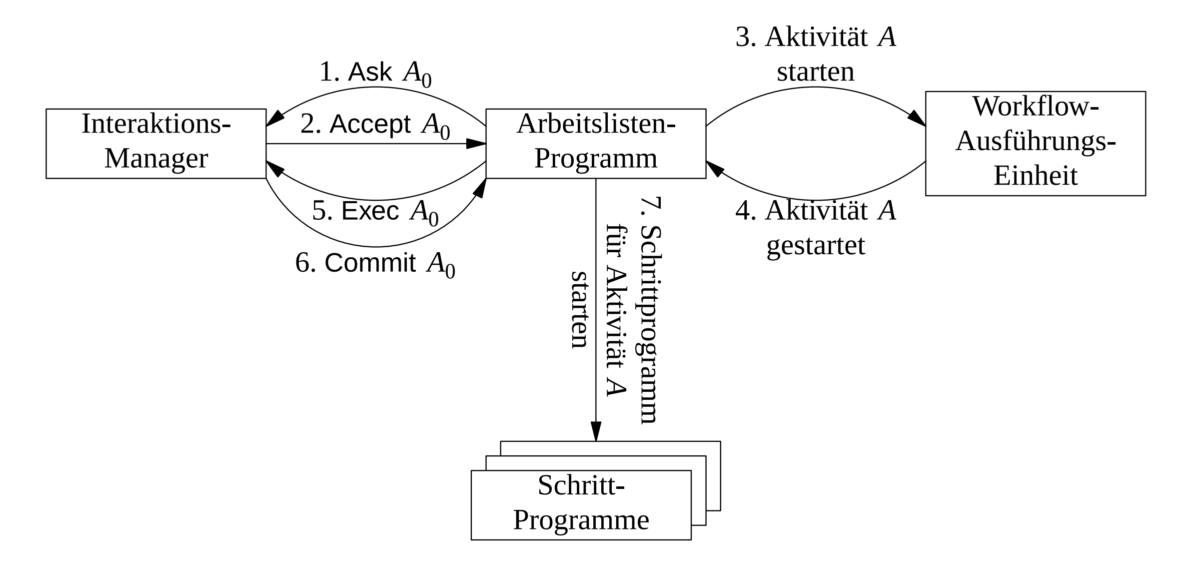


Abbildung 5.42: Starten einer Aktivität

Nachricht für die Aktion A_0, die dazu führt, daß die Aktivität A aus der Arbeitsliste entfernt wird (vgl. Abb. 5.41).

- Falls in Schritt 4 (typischerweise aus demselben Grund wie oben) keine Bestätigung der Workflow-Ausführungseinheit eintrifft, so wird in Schritt 5 keine Exec-, sondern stattdessen eine Undo-Nachricht an den Interaktionsmanager gesandt. Die Schritte 6 und 7 entfallen in diesem Fall.
- Sollte schließlich in Schritt 6 (typischerweise aufgrund unerwarteter Verzögerungen) keine Commit-, sondern eine Abort-Nachricht des Managers eintreffen, so kann das Arbeitslistenprogramm die Schritte 1 bis 6 (mit Ausnahme von Schritt 3 und 4) so lange wiederholen, bis es in Schritt 6 eine Commit-Nachricht erhält. Sollte hierbei jedoch in Schritt 2 eine Reject-Nachricht eintreffen, so muß die Workflow-Ausführungseinheit aufgefordert werden, die Ausführung der Aktivität A abzubrechen.

Anmerkung: Die exakten Details der Schritte 3, 4 und 7 hängen stark vom jeweiligen WfMS ab; das Grundprinzip dürfte jedoch bei allen Systemen ähnlich sein.

5.5.2.3 Beenden von Aktivitäten

Das Beenden einer Aktivität verläuft normalerweise ähnlich wie das Starten (vgl. Abb. 5.43):

- Wenn ein Schrittprogramm einer Aktivität A terminiert (1), sendet das Arbeitslistenprogramm eine Wait-Nachricht für die Endeaktion A_1 an den Interaktionsmanager (2), die normalerweise sofort mit Accept beantwortet wird (3).[19]
- Anschließend wird die Workflow-Ausführungseinheit aufgefordert, die Aktivität zu beenden (4).
- Wird dies von der Ausführungseinheit bestätigt (5), so wird eine Exec-Nachricht an den Interaktionsmanager gesandt (6), die normalerweise mit Commit beantwortet wird (7).

[19] Da eine Aktivität A in einem Interaktionsgraphen durch die Sequenz $A_0 - A_1$ ersetzt wird, ist A_1 im Anschluß an A_0 immer zulässig, es sei denn, es würde im Graphen an irgendeiner Stelle isoliert (d. h. ohne vorangehendes A_0) auftreten.

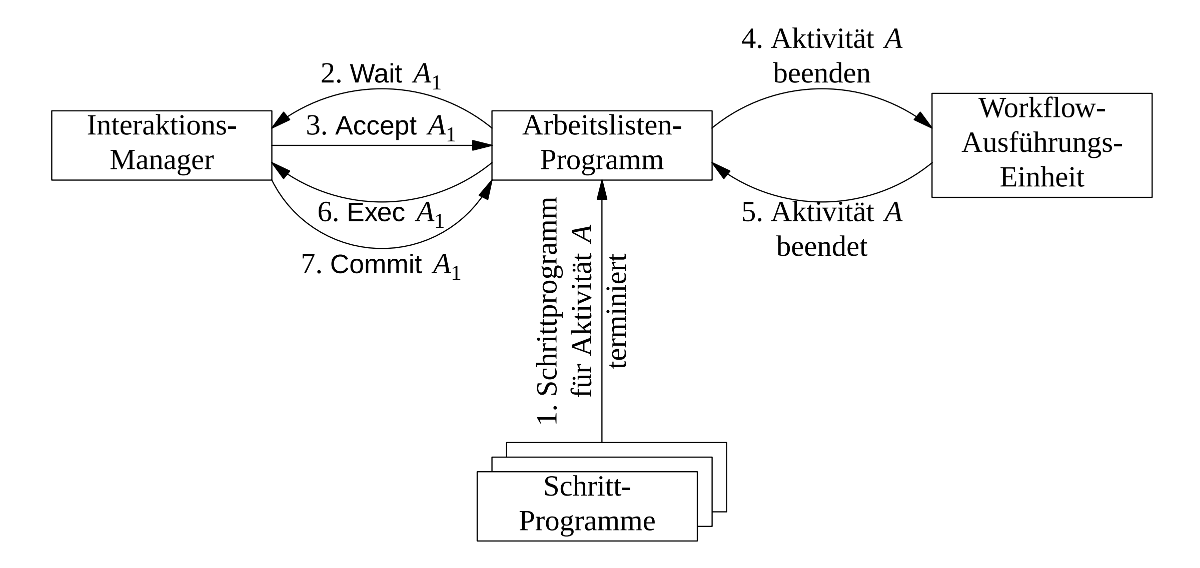


Abbildung 5.43: Beenden einer Aktivität

Auch hier sind jedoch gewisse Abweichungen vom Normalfall zu berücksichtigen:

- Trifft in Schritt 3 anstelle der Accept- eine Unknown-Nachricht ein, so entfallen die Schritte 6 und 7.
- Aufgrund ungewöhnlich formulierter Interaktionsgraphen (vgl. Fußnote) könnte es passieren, daß die Aktion A_1 wider Erwarten unzulässig ist und die Accept-Nachricht daher nicht sofort eintrifft. In diesem Fall verzögern sich die Schritte 4 bis 7, d. h. die Aktivität bleibt aus Sicht des WfMSs aktiv, obwohl das zugehörige Schrittprogramm bereits beendet ist.
- Ebenso wie beim Starten einer Aktivität, kann es aufgrund unerwarteter Verzögerungen vorkommen, daß in Schritt 7 keine Commit-, sondern eine Abort-Nachricht des Interaktionsmanagers eintrifft. In diesem Fall sollte das Arbeitslistenprogramm die Schritte 2 bis 7 (mit Ausnahme von Schritt 4 und 5) so lange wiederholen, bis in Schritt 7 eine Commit-Nachricht eintrifft.

5.5.2.4 Verallgemeinerung auf mehrere Interaktionsmanager und Ausführungseinheiten

Die in den vorangegangenen Abschnitten beschriebenen Schrittfolgen zur Aktualisierung von Arbeitslisten sowie zum Starten und Beenden von Aktivitäten lassen sich ohne weiteres auf Systemarchitekturen verallgemeinern, in denen es mehrere Interaktionsmanager oder Workflow-Ausführungseinheiten gibt (vgl. Abb. 5.44).

In der Arbeitsliste eines Benutzers stehen dann zu jedem Zeitpunkt all jene Aktivitäten, die aus Sicht *irgendeiner* Workflow-Ausführungseinheit und aus Sicht *aller* Interaktionsmanager zulässig sind. Beim Starten und Beenden einer Aktivität A sind jeweils *eine* Ausführungseinheit und *alle* Interaktionsmanager, die die Aktion A_0 bzw. A_1 kennen, involviert. Zur Koordination dieser Manager kann entweder das modifizierte dreiphasige Koordinationsprotokoll (vgl. § 5.2.7.3) oder eventuell auch das zweiphasige Koordinationsprotokoll (vgl. § 5.2.3.1) verwendet werden.

Anmerkung: In § 5.2.7.4 wurde erläutert, daß der Anfragetyp Wait für ein Koordinationsprotokoll mit mehreren Interaktionsmanagern grundsätzlich ungeeignet ist, da die Antwort auf eine solche Anfrage u. U. sehr lange ausbleiben kann. Da die Accept-Antwort für eine Endeaktion A_1 aber gemäß § 5.5.2.3 normalerweise sofort eintrifft, tritt dieses Problem beim Einsatz „normaler“ Graphen nicht auf.

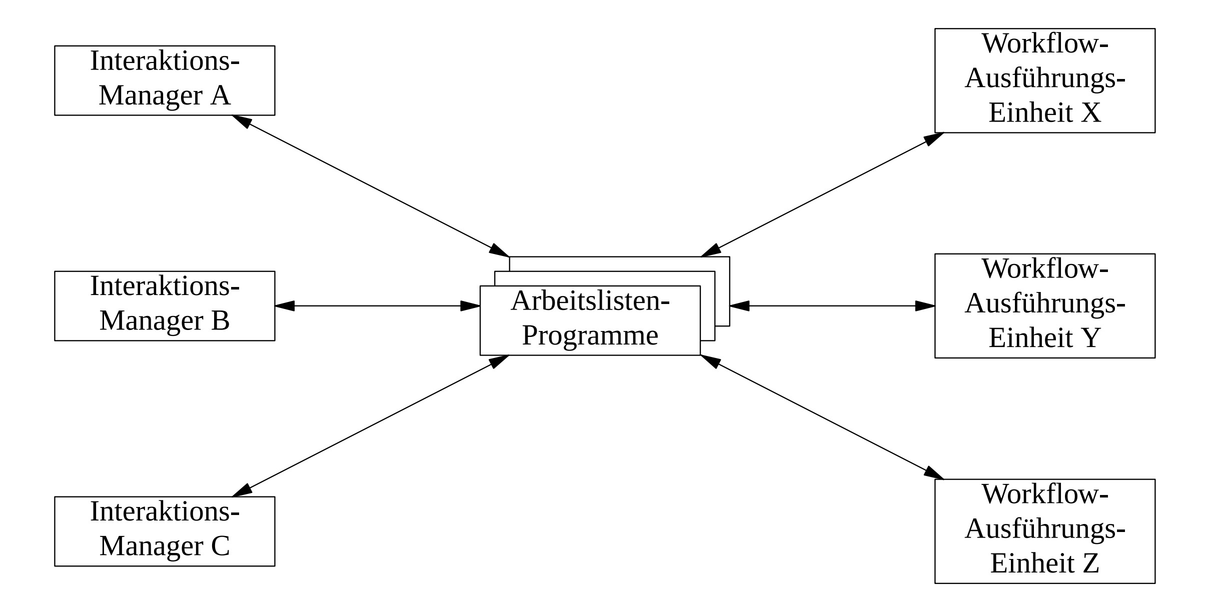


Abbildung 5.44: Mehrere Interaktionsmanager und Workflow-Ausführungseinheiten

5.5.3 Adaption von Workflow-Ausführungseinheiten

5.5.3.1 Aktualisierung von Arbeitslisten

Eine zweite Möglichkeit, Workflow-übergreifende Integritätsbedingungen zu implementieren, besteht darin, die zentrale Ausführungseinheit eines Workflow-Management-Systems geeignet zu modifizieren. Abbildung 5.45 zeigt das resultierende Zusammenspiel von Interaktionsmanager, Workflow-Ausführungseinheit und Arbeitslistenprogrammen zur Aktualisierung von Arbeitslisten:

- Für jede Aktivität A, die aus Sicht der Workflow-Ausführungseinheit zulässig ist, wird eine Subscribe-Nachricht für die Startaktion A_0 an den Interaktionsmanager gesandt (1).
- Erst wenn vom Manager eine Permit-Nachricht für die Aktion A_0 eintrifft (2), wird den Arbeitslistenprogrammen mitgeteilt, daß die Aktivität A zulässig ist (3). Trifft später eine zugehörige Forbid-Nachricht ein (4), wird diese Mitteilung widerrufen (5). Dieses Wechselspiel kann sich beliebig oft wiederholen (2−5).
- Ist eine Aktivität A aus Sicht der Workflow-Ausführungseinheit nicht mehr zulässig, so wird eine Cancel-Nachricht für die Startaktion A_0 an den Interaktionsmanager gesandt (6) und den Arbeitslistenprogrammen mitgeteilt, daß A nicht mehr zulässig ist (7).

Bei den Arbeitslistenprogrammen handelt es sich entweder um die zum WfMS gehörenden Standardprogramme (vgl. § 1.1.6) oder um entsprechende Eigenimplementierungen. Im Gegensatz zu den adaptierten Programmen in § 5.5.2, wissen die hier betrachteten Arbeitslistenprogramme jedoch nichts



Abbildung 5.45: Aktualisierung von Arbeitslisten

von der Existenz des Interaktionsmanagers. Sie reagieren daher auf die Mitteilungen 3, 5 und 7 jeweils ohne weitere Nachfragen, indem sie die genannten Aktivitäten in der Arbeitsliste des Benutzers anzeigen bzw. aus ihr entfernen.

Anmerkung: Die Behandlung von Unknown-Nachrichten des Interaktionsmanagers erfolgt analog zu § 5.5.2.1. Entsprechendes gilt auch für die beiden nachfolgenden Abschnitte 5.5.3.2 und 5.5.3.3.

5.5.3.2 Starten von Aktivitäten

Abbildung 5.46 zeigt das Zusammenspiel der Komponenten beim Starten einer Aktivität. Da der Interaktionsmanager hier mit zuverlässigen Workflow-Ausführungseinheiten und nicht mit potentiell unzuverlässigen Arbeitslistenprogrammen kommuniziert, kann das einfachere zweiphasige Koordinationsprotokoll verwendet werden (vgl. § 5.2.7.3), das normalerweise wie folgt abläuft:

- Wenn ein Benutzer eine Aktivität A starten will, sendet sein Arbeitslistenprogramm eine entsprechende Mitteilung an die Workflow-Ausführungseinheit (1).
- Diese stellt eine Ask-Anfrage für die Startaktion A_0 an den Interaktionsmanager (2) und wartet auf die zugehörige Accept-Antwort (3).
- Nach deren Eintreffen wird die Aktivität intern gestartet (d. h. zum Beispiel eine entsprechende Datenbank-Transaktion ausgeführt), und es werden Bestätigungen an den Interaktionsmanager (4) und das Arbeitslistenprogramm (5) versandt.

Sollte in Schritt 3 wider Erwarten keine Accept-, sondern eine Reject-Antwort eintreffen, so entfällt Schritt 4, während in Schritt 5 eine entsprechende negative Mitteilung an das Arbeitslistenprogramm gesandt wird.

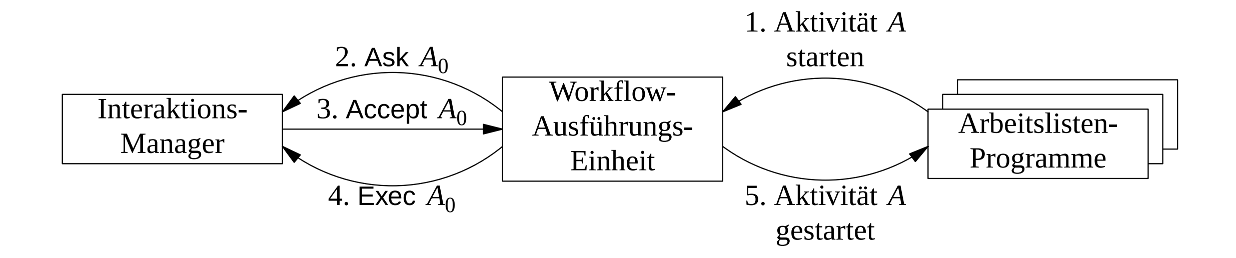


Abbildung 5.46: Starten einer Aktivität

5.5.3.3 Beenden von Aktivitäten

Abbildung 5.47 zeigt schließlich die Schrittfolge zum Beenden einer Aktivität:

- Wenn ein Schrittprogramm einer Aktivität A terminiert, sendet das Arbeitslistenprogramm eine entsprechende Mitteilung an die Workflow-Ausführungseinheit (1).
- Diese stellt eine Wait-Anfrage für die Endeaktion A_1 an den Interaktionsmanager (2) und wartet auf die zugehörige Accept-Antwort (3).
- Nach deren Eintreffen wird die Aktivität intern beendet (wiederum z. B. durch Ausführen einer entsprechenden Datenbank-Transaktion), und es werden Bestätigungen an den Interaktionsmanager (4) und das Arbeitslistenprogramm (5) versandt.

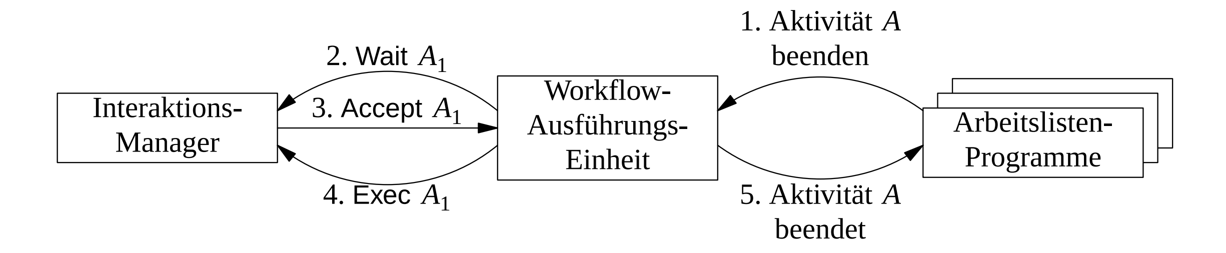


Abbildung 5.47: Beenden einer Aktivität

5.5.3.4 Verallgemeinerung auf mehrere Interaktionsmanager und Ausführungseinheiten

Auch das Prinzip der Adaption von Workflow-Ausführungseinheiten läßt sich auf ein System von mehreren Interaktionsmanagern und Ausführungseinheiten verallgemeinern. Abbildung 5.48 zeigt dies exemplarisch für jeweils drei dieser Komponenten. Während sich die Workflow-Ausführungseinheiten potentiell mit allen Interaktionsmanagern abstimmen müssen, kommuniziert ein Arbeitslistenprogramm jeweils nur mit *einer* Ausführungseinheit und weiß nichts von der Existenz der Interaktionsmanager. Dementsprechend befinden sich in der Arbeitsliste eines Benutzers immer nur Aktivitäten *eines* Workflow-Management-Systems, die aus Sicht dieses Systems und aus Sicht *aller* Interaktionsmanager zulässig sind.

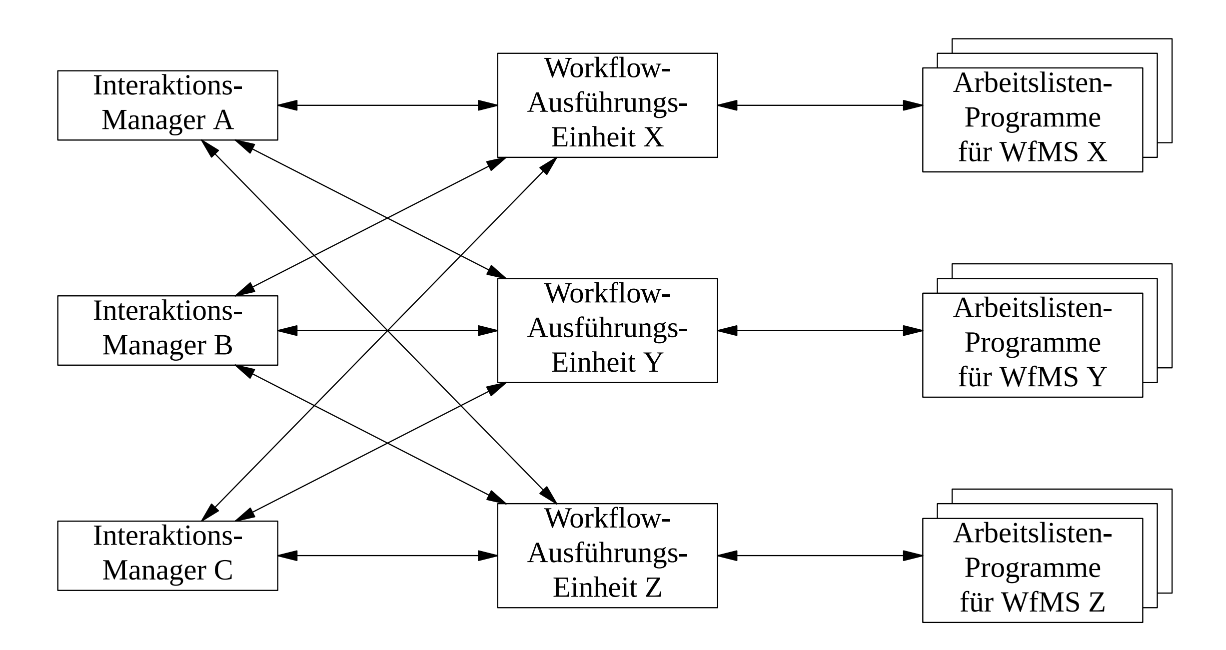


Abbildung 5.48: Mehrere Interaktionsmanager und Workflow-Ausführungseinheiten

Beim Starten und Beenden einer Aktivität A sind − ebenso wie in § 5.5.2.4 − jeweils *eine* Ausführungseinheit und *alle* Interaktionsmanager, die die Aktion A_0 bzw. A_1 kennen, involviert. Zur Koordination dieser Manager wird normalerweise das zweiphasige Koordinationsprotokoll verwendet.

5.5.4 Diskussion

Obwohl die Adaption von Arbeitslistenprogrammen und die Adaption von Workflow-Ausführungseinheiten konzeptionell sehr ähnlich sind, besitzen sie doch unterschiedliche Vor- und Nachteile, die im folgenden erörtert werden sollen.

5.5.4.1 Adaption von Arbeitslistenprogrammen

Die Adaption bzw. Eigenentwicklung eines Arbeitslistenprogramms wird von einem Workflow-Management-System normalerweise explizit durch die Bereitstellung einer entsprechenden Schnittstelle (Nummer 2 im Referenzmodell der WfMC; vgl. Abb. 1.3, § 1.1.6) unterstützt. Obwohl der Programmieraufwand nicht zu vernachlässigen ist, entscheidet man sich in vielen Anwendungen dafür, die vom WfMS mitgelieferten Standardprogramme durch eigene Programme zu ersetzen, die sich besser in den Gesamtkontext einer Arbeitsplatz-Oberfläche einfügen oder eine flexiblere, auf die jeweilige Anwendung zugeschnittene Gestaltung und Präsentation der Arbeitslisten erlauben. So könnte man im Beispiel der medizinischen Untersuchungsworkflows Aktivitäten jeweils nach dem Patienten gruppieren, auf den sie sich beziehen, damit z. B. mehrere Vorbereitungsmaßnahmen für denselben Patienten in der Arbeitsliste unmittelbar hintereinander stehen oder virtuell zu einer einzigen Aktivität verschmolzen werden.[20] Sofern aufgrund derartiger Überlegungen ohnehin spezielle Arbeitslistenprogramme entwickelt werden, können die in § 5.5.2 beschriebenen Adaptionen hierbei ohne große Mühe berücksichtigt werden.

Ein wesentlicher Nachteil dieses Ansatzes besteht jedoch darin, daß Workflow-übergreifende Integritätsbedingungen am eigentlichen Workflow-Management-System „vorbei“ implementiert werden. Da die zentrale Workflow-Ausführungseinheit nichts von der Existenz des Interaktionsmanagers weiß, kann sie selbst nichts zur Einhaltung der Integritätsbedingungen beitragen. Das bedeutet, daß selbst Systemadministratoren nur die adaptierten Arbeitslistenprogramme zur Kommunikation mit dem WfMS verwenden sollten, da bei Verwendung der Standardprogramme die erforderliche Abstimmung mit dem Interaktionsmanager fehlt.

Ein weiterer Nachteil besteht darin, daß normalerweise sehr viele Arbeitslistenprogramme gleichzeitig aktiv sind, die alle mit dem Interaktionsmanager kommunizieren müssen. Wird eine Aktivität A, die aus Sicht der Workflow-Ausführungseinheit zulässig ist, beispielsweise an 100 aktive Arbeitslistenprogramme übermittelt (Schritt 1 in Abb. 5.41), so senden diese alle eine Subscribe-Nachricht für die Startaktion A_0 an den Manager. Ist die Aktion zulässig, so muß dieser wiederum 100 Permit-Nachrichten verschicken usw. Im Gegensatz hierzu erhält der Manager bei der Adaption von Ausführungseinheiten (Abb. 5.45) nur *eine* Subscribe-Nachricht und muß entsprechend nur *eine* Permit-Antwort verschicken.

Schließlich sind Arbeitslistenprogramme (bzw. die Rechner, auf denen sie ausgeführt werden), wie bereits erwähnt, potentiell unzuverlässig, was den Einsatz des relativ aufwendigen dreiphasigen Koordinationsprotokolls erfordert.

Trotz dieser Nachteile ist die Adaption von Arbeitslistenprogrammen insofern interessant, als sie − wie oben erwähnt − mit relativ geringem Zusatzaufwand realisiert werden kann. Außerdem stellt sie die einzige Möglichkeit dar, Inter-Workflow-Abhängigkeiten auf *Anwendungsebene*, d. h. ohne Eingriff in ein WfMS, zu implementieren.

5.5.4.2 Adaption von Workflow-Ausführungseinheiten

Die Adaption von Workflow-Ausführungseinheiten besitzt den entscheidenden Vorteil, daß Inter-Workflow-Abhängigkeiten als „first class citizens“ behandelt werden, deren Einhaltung in der gleichen Weise vom WfMS garantiert wird wie die Einhaltung gewöhnlicher Intra-Workflow-Abhängigkeiten. Dies bedeutet insbesondere, daß ihre Existenz für Arbeitslistenprogramme − egal ob vom WfMS vordefiniert oder selbst implementiert − transparent ist. Auf diese Weise ist sichergestellt, daß

[20] Bei der Ausführung einer solchen virtuellen Aktivität durch den Benutzer könnte das Arbeitslistenprogramm automatisch mehrere echte Aktivitäten auf einmal starten.

Workflow-übergreifende Integritätsbedingungen weder absichtlich noch versehentlich durch die Verwendung eines nicht adaptierten Arbeitslistenprogramms verletzt werden können.

Vergleicht man Inter-Workflow-Abhängigkeiten wieder mit Integritätsbedingungen in Datenbanken (vgl. § 5.4.3), so entspricht dieser Lösungsansatz der Spezifikation und Implementierung von Integritätsbedingungen direkt im Datenbankkern, während die Adaption von Arbeitslistenprogrammen einer Verlagerung der Bedingungen in die einzelnen Datenbank-Anwendungen entspricht. Abgesehen von der größeren Fehleranfälligkeit, muß hierbei auch der Programmcode zur Implementierung der Bedingungen in sämtlichen Anwendungen (d. h. hier eventuell verschiedenen Arbeitslistenprogrammen) repliziert werden.

Ungeachtet dieser Vorteile, besitzt die Adaption von Ausführungseinheiten den Nachteil, daß sie implementierungstechnisch wesentlich anspruchsvoller und schwieriger zu realisieren ist als die Adaption von Arbeitslistenprogrammen. Da eine Workflow-Ausführungseinheit ein umfangreiches und komplexes Software-System darstellt, erscheint eine *nachträgliche* Adaption kaum möglich. Vielmehr muß die Existenz von Inter-Workflow-Abhängigkeiten bzw. eines Interaktionsmanagers, der sie implementiert, bereits in der Entwurfs- und Entwicklungsphase einer Ausführungseinheit angemessen berücksichtigt werden.

5.5.5 Beispiel

Um die Ausführungen der Abschnitte 5.5.2 und 5.5.3 etwas plastischer werden zu lassen, soll im folgenden ein typisches Ablaufszenario, bestehend aus zwei Untersuchungs-Workflows und der Integritätsbedingung für Patienten, Schritt für Schritt durchgespielt werden. Obwohl das Beispiel konkret anhand adaptierter Arbeitslistenprogramme (§ 5.5.2) vorgestellt wird, kann es nahezu unverändert auf die Adaption von Workflow-Ausführungseinheiten (§ 5.5.3) übertragen werden.

Angenommen, ein Patient X klagt über immerwiederkehrende Bauchschmerzen und Verdauungsbeschwerden und wird daher von seinem Hausarzt zur Abklärung der Ursache in eine internistische Klinik überwiesen. Nach seiner administrativen Aufnahme wird ihm ein Bett auf einer Station zugewiesen.

Am nächsten Morgen kommt ein Stationsarzt zur Visite und ordnet eine Oberbauchsonographie sowie eine Darmspiegelung (Endoskopie) an, d. h. er startet einen Sonographie- und einen Endoskopie-Workflow für den Patienten X (vgl. Abb. 5.35 und 5.36, § 5.4.4). Da die initialen Aktivitäten[21] anordnen(X, sono) und anordnen(X, endo) dieser Workflows in der Integritätsbedingung für Patienten (Abb. 5.32, § 5.4.3) nicht vorkommen, können sie ohne Rücksprache mit dem Interaktionsmanager sofort ausgeführt werden. Dasselbe gilt für die anschließend aus Sicht des WfMSs zulässigen Aktivitäten vereinbaren(X, sono) und vereinbaren(X, endo).

Die aus Sicht des WfMSs ebenfalls zulässigen Aktivitäten vorbereiten(X, sono), vorbereiten(X, endo) und aufklären(X, endo) kommen jedoch im betrachteten Interaktionsgraphen vor und dürfen daher nur ausgeführt bzw. zur Ausführung angeboten werden, wenn dies vom Interaktionsmanager genehmigt wird. Da ihre Startaktionen im initialen Zustand des Graphen alle zulässig sind, werden entsprechende Subscribe-Anfragen der Arbeitslistenprogramme jeweils mit Permit-Nachrichten beantwortet, so daß die Aktivitäten tatsächlich in den Arbeitslisten der autorisierten Benutzer erscheinen.

Abbildung 5.49 zeigt den momentanen Verarbeitungszustand. Eine graue Marke in der linken Hälfte einer Aktivität A des Graphen zeigt an, daß die Startaktion A_0 dieser Aktivität im momentanen Zustand des Interaktionsgraphen zulässig ist, d. h. daß A aus Sicht des Interaktionsmanagers gestartet werden darf.

[21] Aus Gründen der Übersichtlichkeit werden die vollständigen Aktivitätenbezeichnungen Untersuchung anordnen, Termin vereinbaren usw. zum Teil durch die zugehörigen Tätigkeitswörter anordnen, vereinbaren usw. abgekürzt.

Aktivitäten eines Workflows, die mit einem Haken versehen sind, wurden bereits ausgeführt, während eingerahmte Aktivitäten sowohl aus Sicht des WfMSs als auch aus Sicht des Interaktionsmanagers zulässig sind und somit in den Arbeitslisten der Benutzer erscheinen.

Sobald eine Stationsschwester anhand ihrer Arbeitsliste erkennt, daß Patient X für die Untersuchungen sono und endo vorbereitet werden muß, beginnt sie, diese beiden Aktivitäten zu starten. Hierbei laufen jeweils die in Abb. 5.42 dargestellten Protokollschritte ab, in deren Verlauf der Interaktionsmanager jeweils einen Zustandsübergang für die Startaktionen vorbereiten_0(X, sono) und vorbereiten_0(X, endo) durchführt. In der Folge akzeptiert der betrachtete Graph (für diesen Patienten) nur noch die zugehörigen Endeaktionen vorbereiten_1(X, sono) und vorbereiten_1(X, endo) sowie weitere vorbereiten_0-Aktionen. Ersteres wird in Abb. 5.50 durch die beiden Marken in der rechten Hälfte der Aktivität vorbereiten angezeigt, während letzteres wie zuvor durch die Marke in der linken Hälfte angezeigt wird.

Da die Startaktion aufklären_0(X, endo) nun nicht mehr zulässig ist, sendet der Interaktionsmanager eine entsprechende Forbid-Nachricht an die Arbeitslistenprogramme der Stationsärzte, die dazu führt, daß die Aktivität aufklären(X, endo) aus den Arbeitslisten entfernt bzw. als momentan nicht ausführbar gekennzeichnet wird. In Abb. 5.50 ist diese Aktivität daher mit abgeschwächter Schrift dargestellt und mit einem gestrichelten Rahmen umgeben, der anzeigt, daß sie zwar aus Sicht des WfMSs, nicht jedoch aus Sicht des Interaktionsmanagers zulässig ist. Die beiden grau unterlegten vorbereiten-Aktivitäten befinden sich gerade in Ausführung.

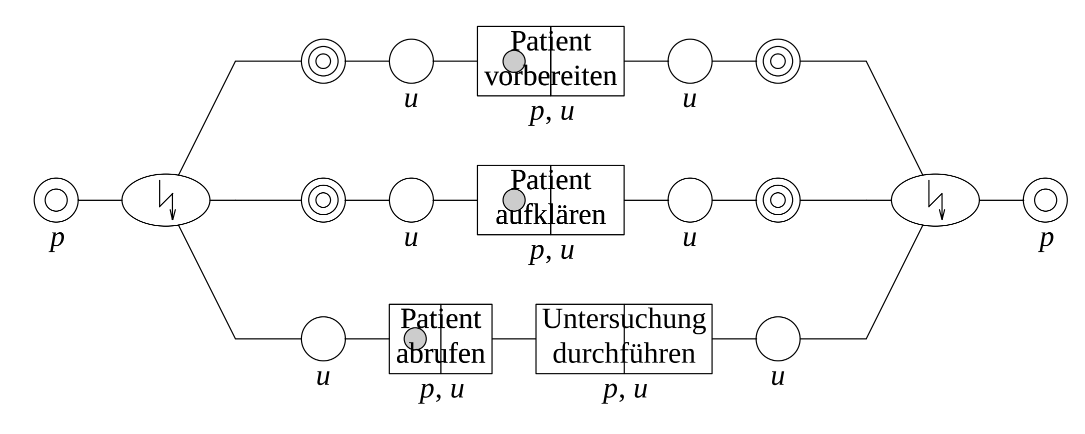
Sobald die Schwester mit der Vorbereitung des Patienten fertig ist, beendet sie die beiden vorbereiten-Aktivitäten gemäß Abb. 5.43. Hierbei führt der Interaktionsmanager Zustandsübergänge für die Endeaktionen vorbereiten_1(X, sono) und vorbereiten_1(X, endo) durch, in deren Folge sich der Graph wieder in seinem initialen Zustand befindet (vgl. Abb. 5.51). Da die Aktion aufklären_0(X, endo) nun wieder zulässig ist, sendet der Interaktionsmanager entsprechende Permit-Nachrichten an die betroffenen Arbeitslistenprogramme, die dazu führen, daß die Aktivität aufklären(X, endo) wieder in den Arbeitslisten der Stationsärzte erscheint.

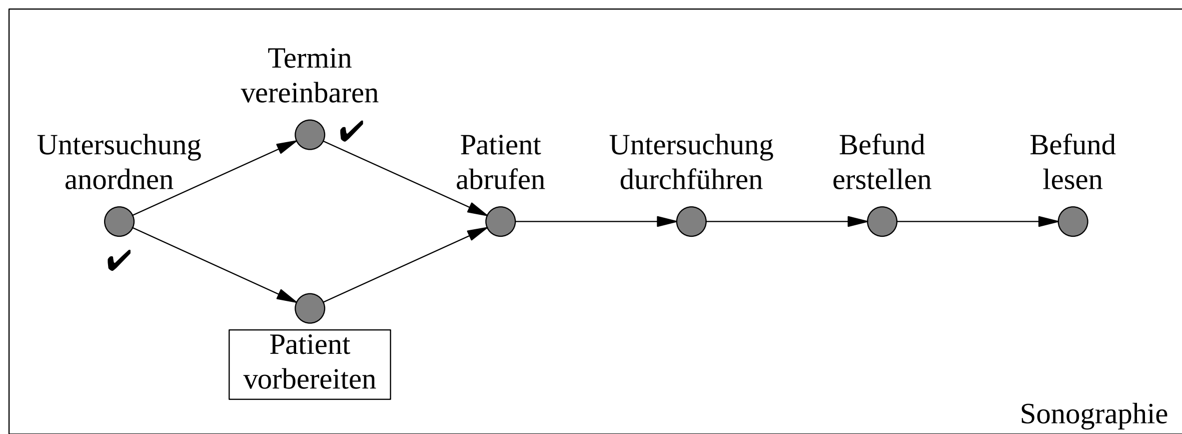
Außerdem steht aus Sicht des WfMSs jetzt die Aktivität abrufen(X, sono) zur Ausführung an, die auch aus Sicht des Interaktionsmanagers zulässig ist, und daher in den Arbeitslisten der Sonographie-MTAs erscheint.

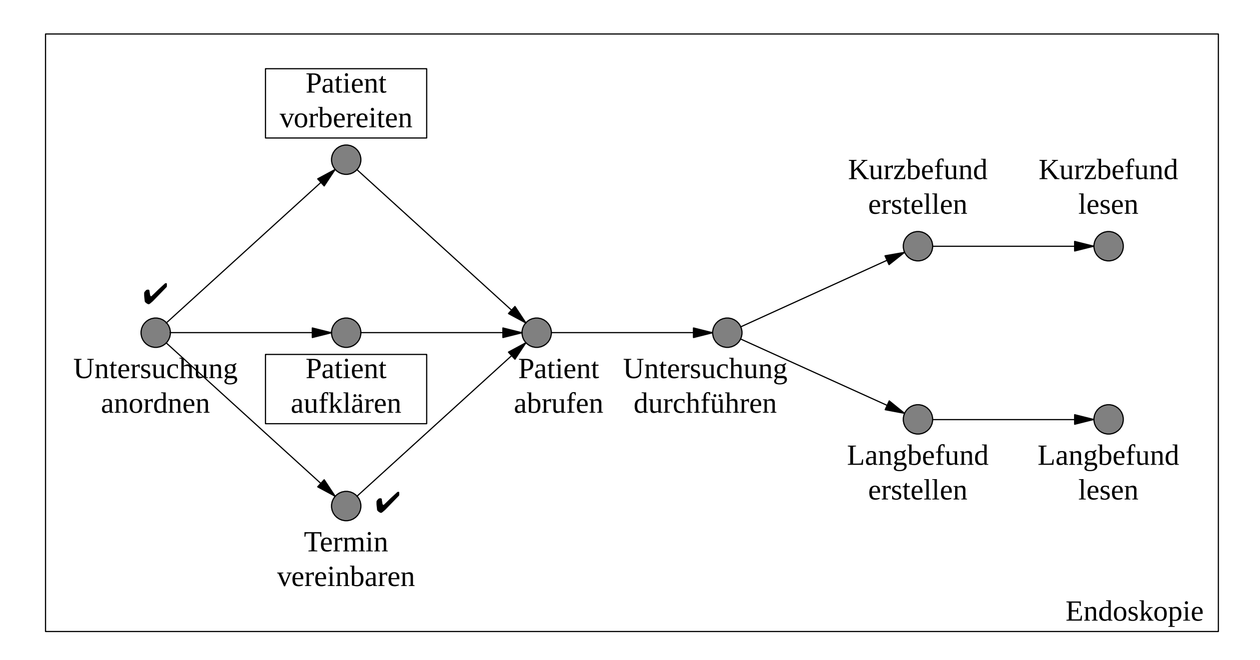
Am Nachmittag kommt ein Stationsarzt, um den Patienten über mögliche Risiken der Darmspiegelung aufzuklären, d. h. er startet die Aktivität aufklären(X, endo), was zur Folge hat, daß der Interaktionsmanager einen Zustandsübergang für die Startaktion aufklären_0(X, endo) durchführt. Abbildung 5.52 zeigt den resultierenden Zustand, in dem der Graph einerseits die Endeaktion aufklären_1(X, endo) und andererseits weitere Startaktionen aufklären_0 erlaubt. Da die Aktion abrufen_0(X, sono) nun nicht mehr zulässig ist, sorgt eine entsprechende Forbid-Nachricht des Interaktionsmanagers dafür, daß die Aktivität abrufen(X, sono) vorübergehend aus den Arbeitslisten verschwindet.

Nach Beendigung des Aufklärungsgesprächs wird ein Zustandsübergang für die Aktion aufklären_1(X, endo) durchgeführt, der zur Folge hat, daß sich der Graph wieder in seinem initialen Zustand befindet und somit die Ausführung von abrufen_0(X, sono) wieder erlaubt. Eine entsprechende Permit-Nachricht des Interaktionsmanagers sorgt daher dafür, daß die Aktivität abrufen(X, sono) wieder in den Arbeitslisten der Sonographie-MTAs erscheint (vgl. Abb. 5.53). Außerdem ist jetzt sowohl aus Sicht des WfMSs als auch aus Sicht des Interaktionsmanagers die Aktivität abrufen(X, endo) zulässig, die somit in den Arbeitslisten der Endoskopie-MTAs erscheint.

Am nächsten Tag wird der Patient als erstes zur Sonographie gerufen, indem eine Sonographie-MTA die Aktivität abrufen(X, sono), d. h. die Aktionen abrufen_0(X, sono) und abrufen_1(X, sono) ausführt. Abbildung 5.54 zeigt den resultierenden Zustand des Graphen sowie die Tatsache, daß die Aktivität abrufen(X, endo) nun vorübergehend nicht ausgeführt werden darf.

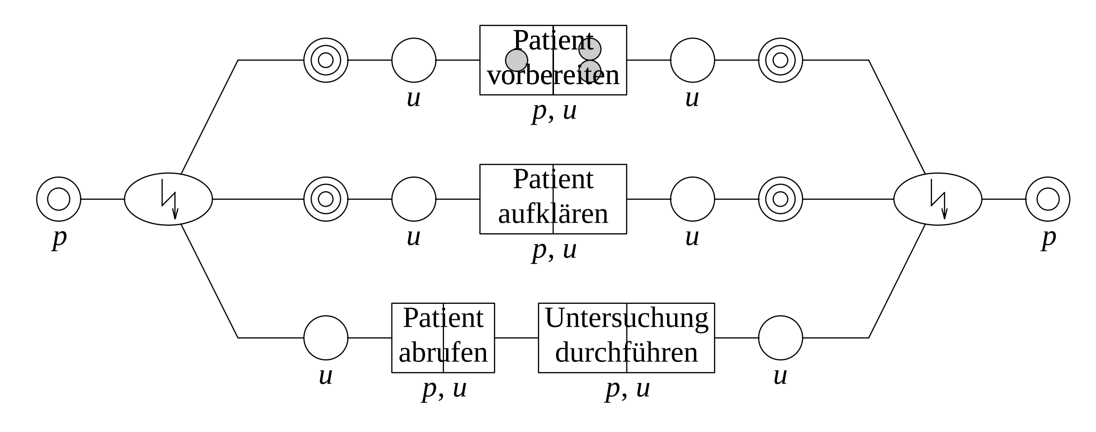


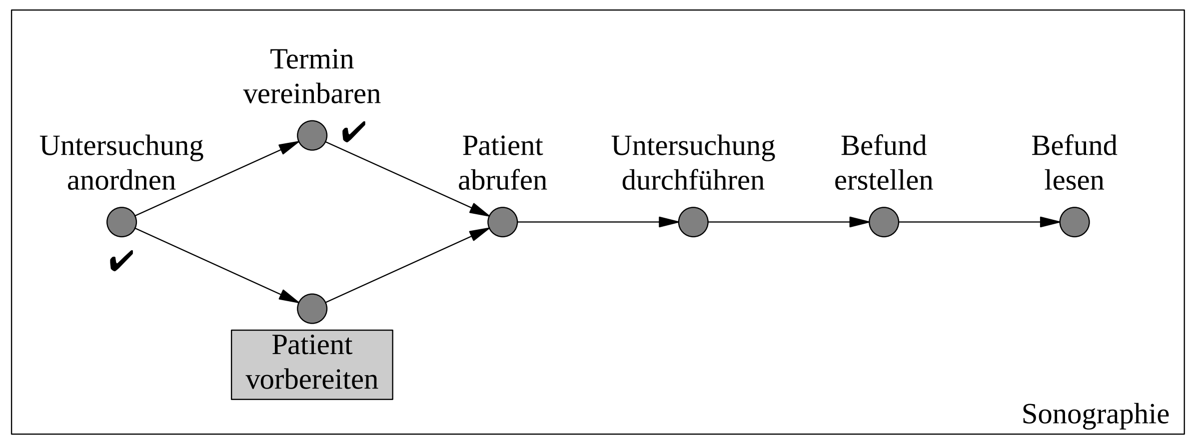


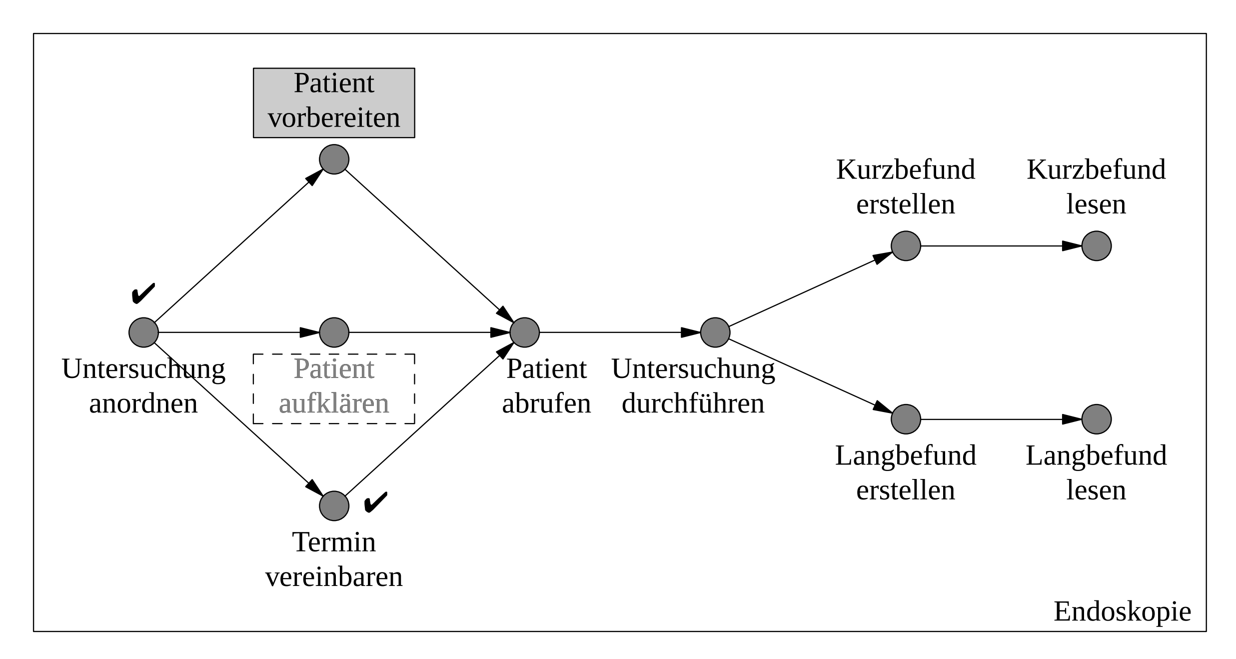


Arbeitslisten der Benutzer					
Station		Sonographie		Endoskopie	
Schwester	Arzt	MTA	Arzt	MTA	Arzt
vorbereiten(X, sono) vorbereiten(X, endo)	aufklären(X, endo)				

Abbildung 5.49: Zustand nach Anordnung der Untersuchungen und Vereinbarung der Termine

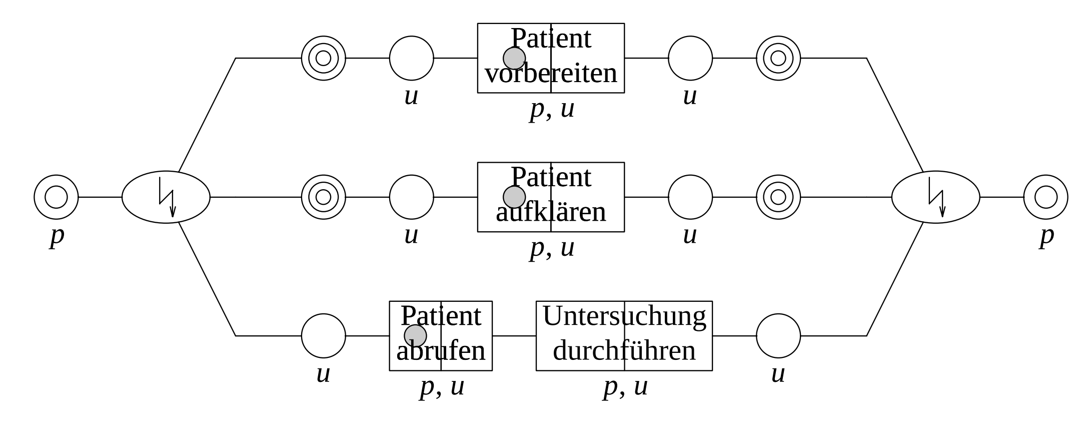


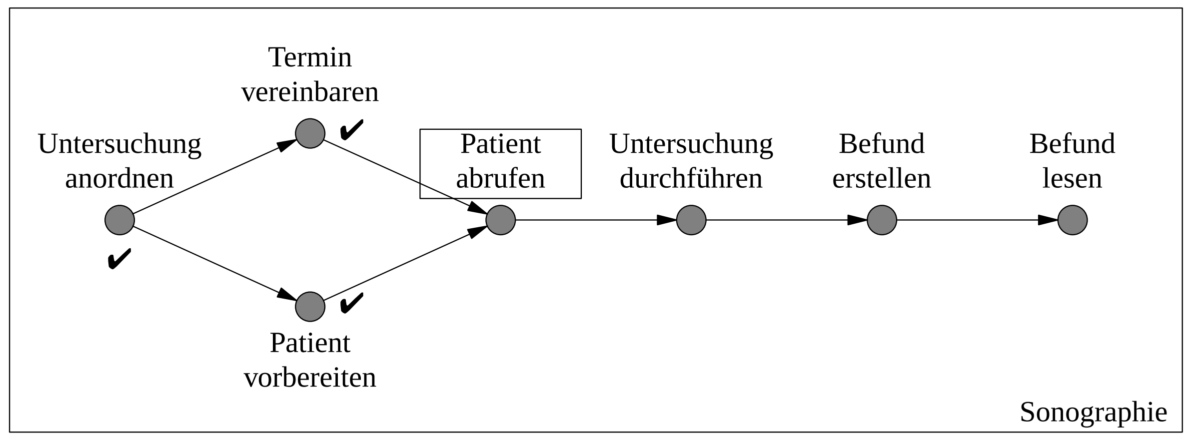


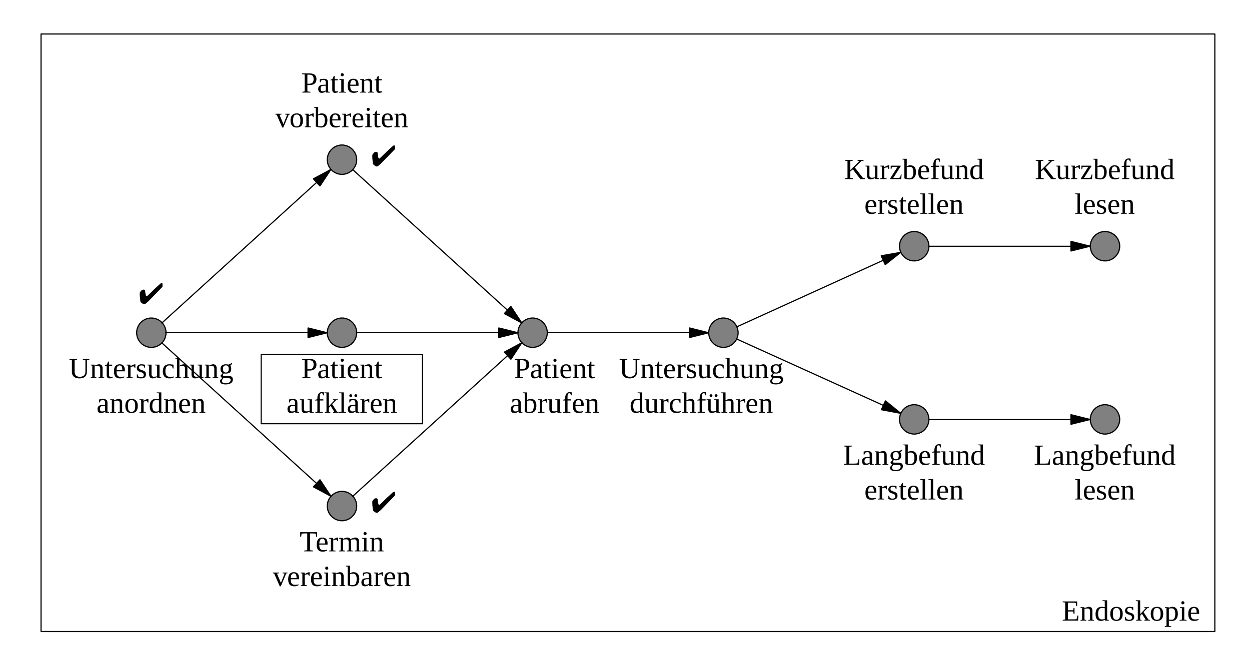


Arbeitslisten der Benutzer					
Station		Sonographie		Endoskopie	
Schwester	Arzt	MTA	Arzt	MTA	Arzt
	aufklären(X, endo)				

Abbildung 5.50: Zustand während der Vorbereitung des Patienten

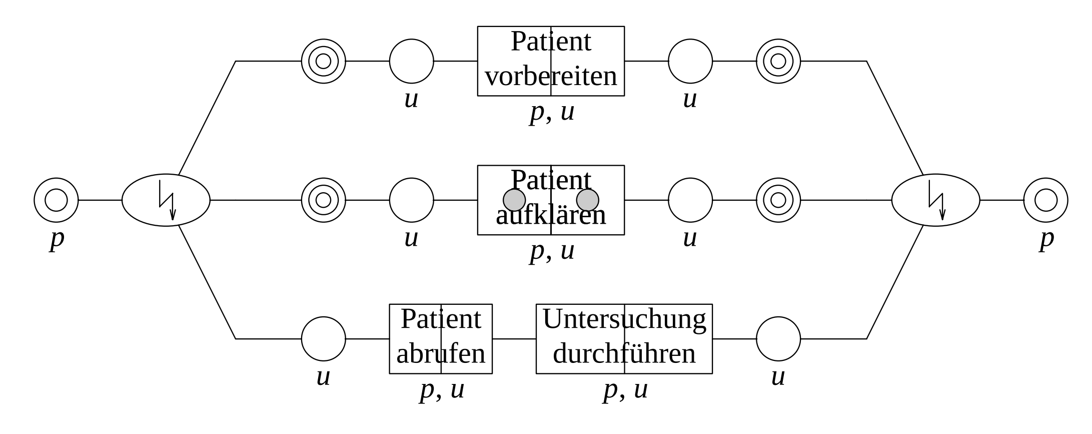


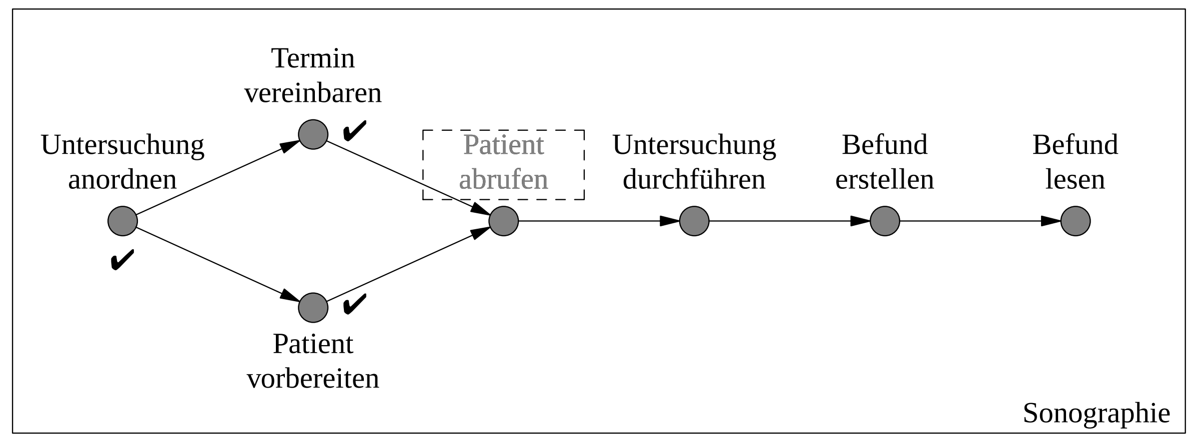


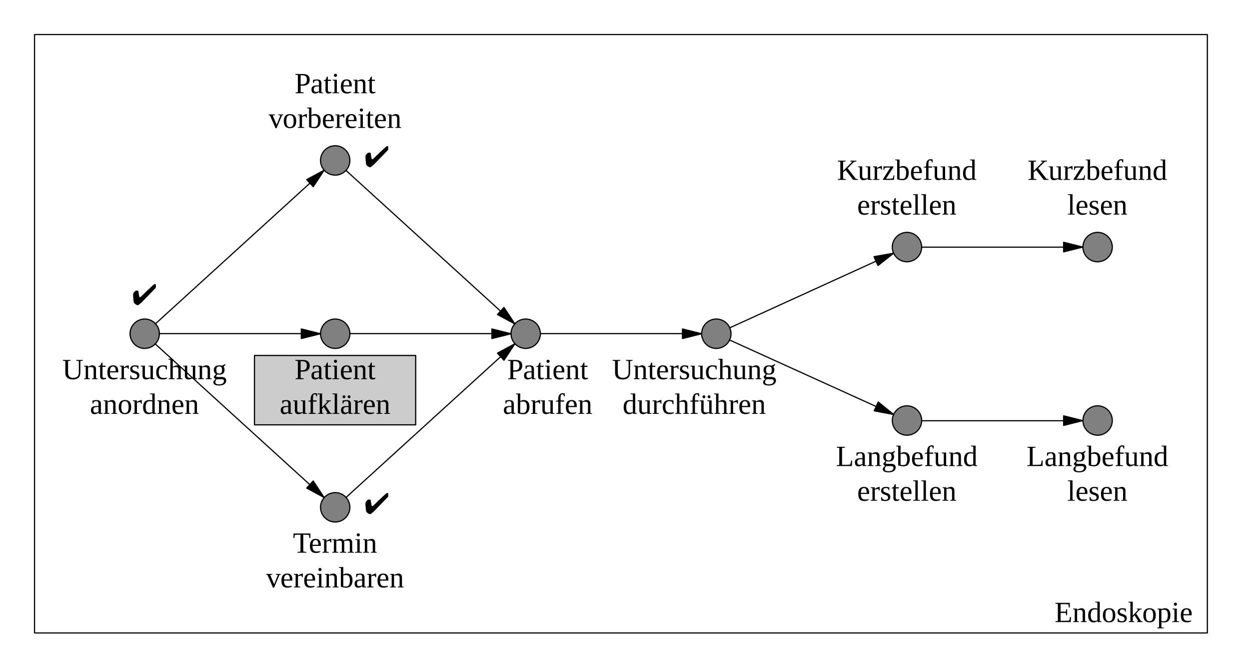


Arbeitslisten der Benutzer					
Station		Sonographie		Endoskopie	
Schwester	Arzt	MTA	Arzt	MTA	Arzt
	aufklären(X, endo)	abrufen(X, sono)			

Abbildung 5.51: Zustand nach Beendigung der Vorbereitung

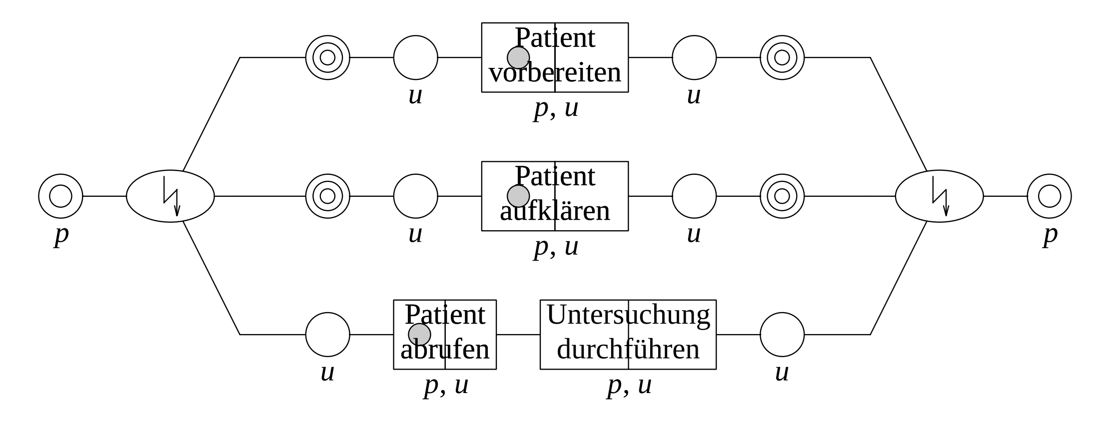


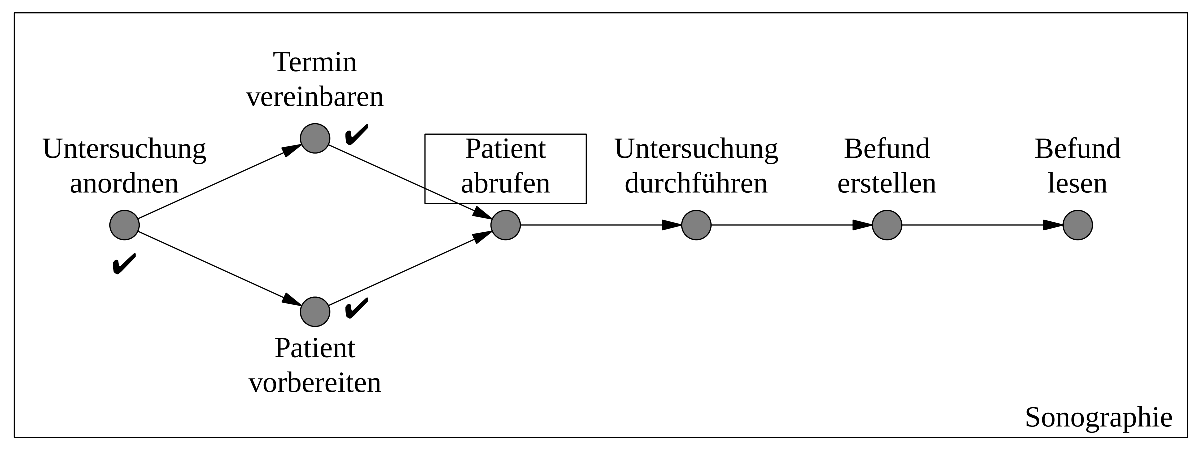


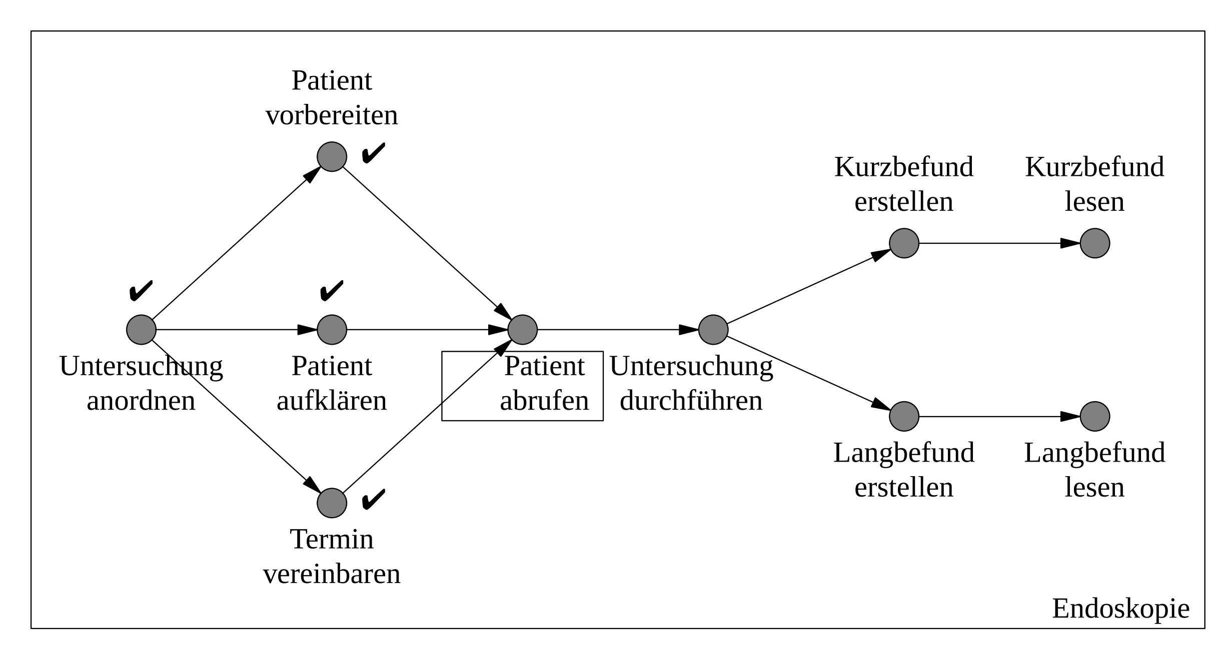


Arbeitslisten der Benutzer						
Station		Sonographie		Endoskopie	
Schwester	Arzt	MTA	Arzt	MTA	Arzt
		abrufen(X, sono)			

Abbildung 5.52: Zustand während der Aufklärung des Patienten

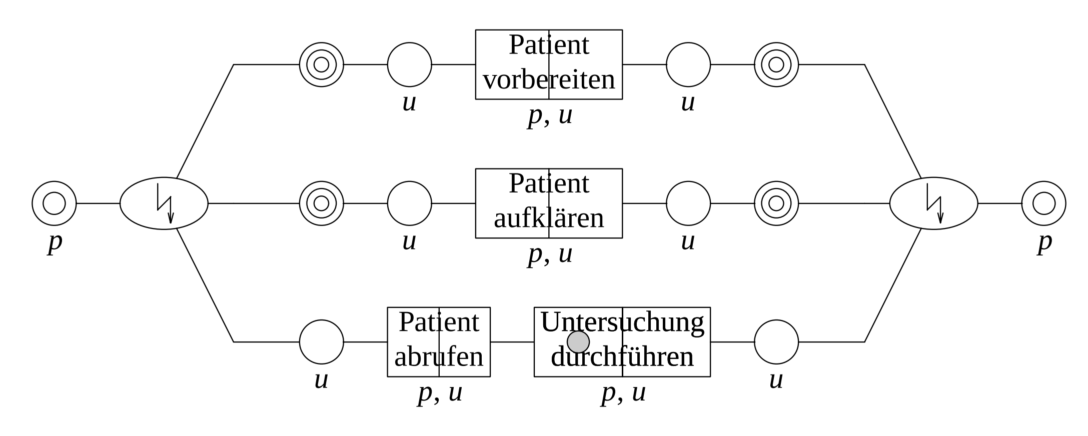


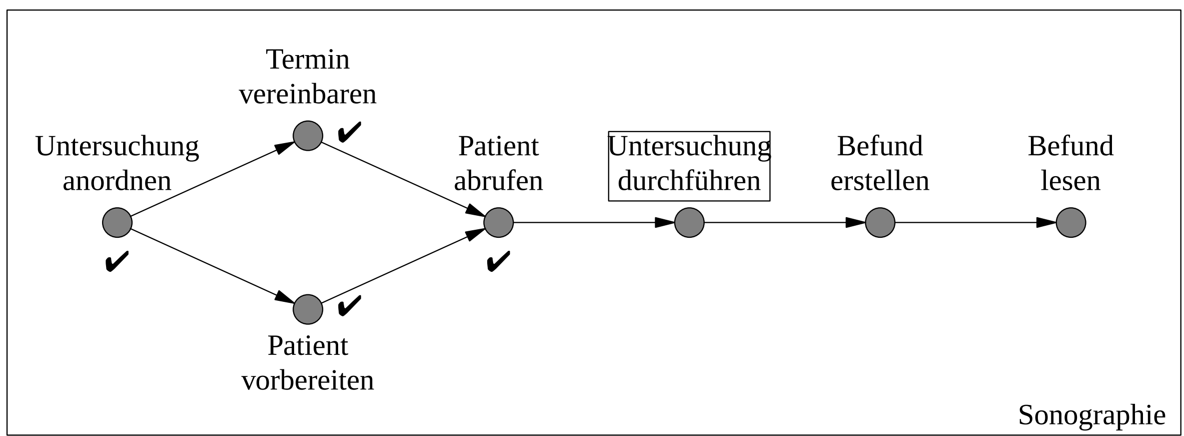


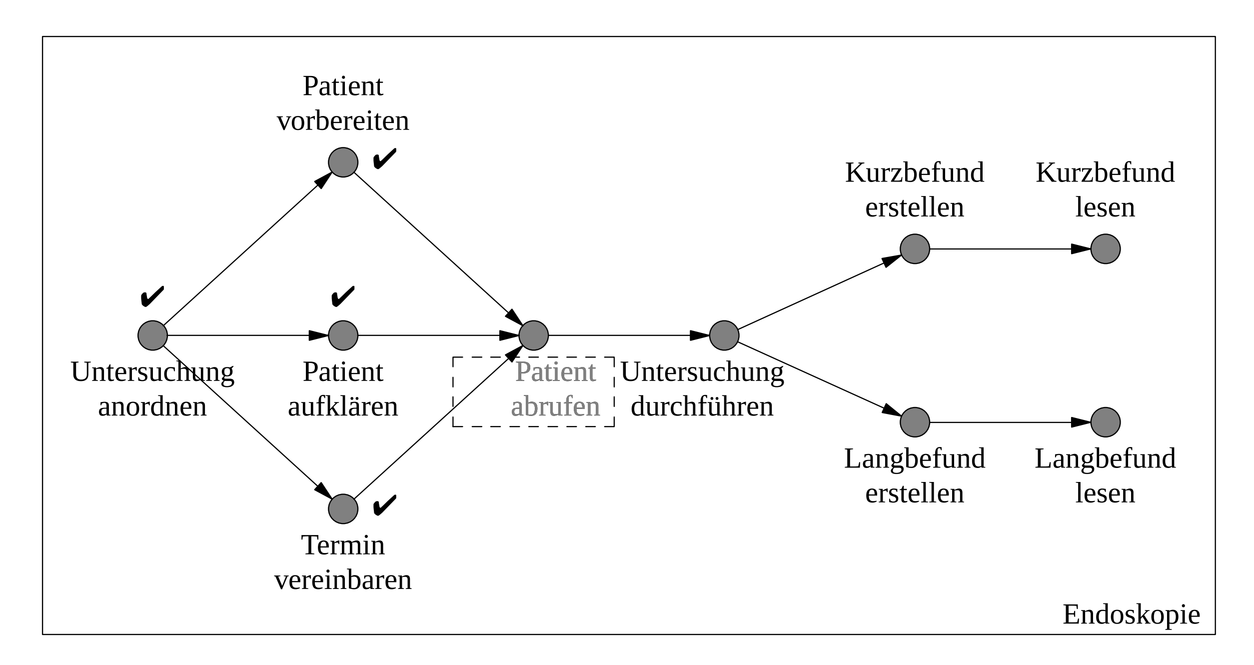


Arbeitslisten der Benutzer					
Station		Sonographie		Endoskopie	
Schwester	Arzt	MTA	Arzt	MTA	Arzt
		abrufen(X, sono)		abrufen(X, endo)	

Abbildung 5.53: Zustand nach Beendigung der Aufklärung







Arbeitslisten der Benutzer					
Station		Sonographie		Endoskopie	
Schwester	Arzt	MTA	Arzt	MTA	Arzt
			untersuchen(X, sono)	abrufen(X, endo)	

Abbildung 5.54: Zustand nach Abruf zur Sonographie

Der weitere Verlauf des Beispiels ist nun offensichtlich. Die einzige ausführbare Aktivität ist momentan untersuchen(X, sono), und nach deren Durchführung können abrufen(X, endo) und untersuchen(X, endo) ausgeführt werden. Da die Aktivitäten (Kurz-/Lang-) Befund erstellen/lesen im betrachteten Interaktionsgraphen nicht vorkommen, können sie nach Belieben überlappend ausgeführt werden, sobald sie aus Sicht des jeweiligen Workflows zulässig sind.

Als mögliche Erweiterung des Beispiels könnte man jedoch annehmen, daß zu einem beliebigen Zeitpunkt weitere Untersuchungen für den Patienten angeordnet werden. In diesem Fall fügen sich die zugehörigen Workflows automatisch in das vorhandene Workflow-Geflecht ein, ebenso wie normal terminierende Workflows das Geflecht wieder verlassen.

Allerdings können Workflows, die abnormal beendet werden, z. B. indem sie vorzeitig abgebrochen werden, u. U. Probleme verursachen. Wenn der Sonographie-Workflow beispielsweise im obigen Zustand abgebrochen wird (weil sich z. B. herausstellt, daß der Patient nicht mehr nüchtern ist und die Untersuchung somit nicht durchgeführt werden kann), so bleibt der Interaktionsgraph im Zustand von Abb. 5.54 „stecken“. Dies hätte zur Folge, daß für den Patienten anschließend keine der Aktivitäten vorbereiten, aufklären oder abrufen mehr ausgeführt werden könnte. Auf diese Problematik − und einen möglichen Lösungsansatz − wird in § 7.2.1.1 etwas näher eingegangen.

Kapitel 6

Verwandte Arbeiten

6.1 Einleitung

6.1.1 Synchronisation paralleler Prozesse

Das Grundproblem der Synchronisation paralleler Prozesse in Computersystemen ist vermutlich schon älter als der Verfasser dieser Arbeit (Jahrgang 1965), und im Laufe der Jahrzehnte wurden, insbesondere im Bereich der Betriebssysteme und Programmiersprachen, zahllose Lösungsvorschläge veröffentlicht und − zum Teil auch sehr kontrovers − diskutiert. Gemäß [Freisleben87] kann das Spektrum der vorgeschlagenen Mechanismen grob in drei Ebenen unterteilt werden:

1. Mechanismen auf der *Hardware-Ebene*, wie z. B. *test-and-set-* oder *decrement-and-test*-Anweisungen, deren Unteilbarkeit durch die Hardware sichergestellt werden muß.

2. *Einfache Mechanismen*, wie z. B. verschiedene Arten von *Semaphoroperationen*, mit deren Hilfe zwar prinzipiell beliebige Synchronisationsprobleme gelöst werden können, deren ausschließliche Verwendung aber schnell zu komplizierten, unübersichtlichen und schwer pflegbaren Formulierungen führt.

3. *Strukturierte Mechanismen*, wie z. B. *kritische Regionen*, *Monitore* oder *Pfadausdrücke*, mit deren Hilfe sich viele Probleme wesentlich kompakter, übersichtlicher und änderungsfreundlicher lösen lassen, als dies mit den „Assembler-Konstrukten“ der Ebene 2 möglich ist.

Typischerweise werden Mechanismen der Ebene 3 auf solche der Ebene 2 (meist Semaphoroperationen) zurückgeführt und diese wiederum mit Hilfe von Hardware-Instruktionen der Ebene 1 implementiert.

6.1.2 Überblick

Da Interaktionsausdrücke und -graphen zweifellos der dritten Ebene zuzuordnen sind und ihre Entwicklung auch durch Vertreter dieser Kategorie (vornehmlich Pfad- und Synchronisierungsausdrücke) inspiriert wurde, sollen im folgenden (§ 6.2) zunächst die Gemeinsamkeiten und Unterschiede zwischen Interaktionsausdrücken und derartigen, auf regulären Ausdrücken basierenden Ansätzen aufgezeigt werden. Hierbei soll u. a. deutlich werden, daß Interaktionsausdrücke in vielen Aspekten über die bisher vorgeschlagenen Formalismen hinausgehen und daher als *Vereinigung und Erweiterung* der meisten existierenden Ansätze betrachtet werden können.[1]

Im weiteren Verlauf des Kapitels werden dann ein prominenter Vertreter von *Prozeßalgebren* (§ 6.3) sowie *erweiterte Transaktionsmodelle* (§ 6.4) diskutiert, bei denen die Verwandtschaft zu Interaktionsausdrücken nicht mehr so offensichtlich ist und z. T. nur noch auf der „terminologischen Ebene“ existiert (d. h. es werden zwar Begriffe wie z. B. *inter-task dependencies* verwendet, die jedoch kaum mit den in dieser Arbeit betrachteten *Inter-Workflow-Abhängigkeiten* vergleichbar sind; vgl. hierzu § 6.4.1.4). Auch die Verwandtschaft von Interaktionsausdrücken oder -graphen zu *Petrinetzen*, auf die in § 6.5 näher eingegangen wird, ist bei weitem nicht so ausgeprägt, wie man auf den ersten Blick − insbesondere wenn man Interaktions*graphen* betrachtet − vermuten könnte.

Zum Abschluß des Kapitels (§ 6.6) folgen einige grundsätzliche Bemerkungen zum Verhältnis zwischen Interaktionsausdrücken und *Workflow-Management-Konzepten* bzw. -Systemen.

[1] Lediglich *ein* wesentliches Konzept von Ereignis- und Flußausdrücken wurde bewußt *nicht* übernommen; vgl. hierzu § 6.2.3.4.

6.2 Erweiterte reguläre Ausdrücke

Die grundsätzliche Idee bei der Entwicklung von Interaktionsausdrücken, *erweiterte reguläre Ausdrücke* zur Beschreibung *zulässiger Ausführungsreihenfolgen* zu verwenden, ist keineswegs neu. Bereits Mitte der 70er und Anfang der 80er Jahre wurden hierfür verschiedene Varianten von *Pfadausdrücken* (§ 6.2.1) vorgeschlagen, und in den 90er Jahren wurde dieselbe Idee in Form von *Synchronisierungsausdrücken* (§ 6.2.2) wiederaufgegriffen. Während diese beiden Formalismen primär als Synchronisationsmechanismen für parallele Programmiersprachen dienen, wurden *Ereignis-* und *Flußausdrücke* (§ 6.2.3) hauptsächlich als Software-Spezifikationssprachen konzipiert. Ähnliches gilt für *CoCoA-Ausführungsregeln* (§ 6.2.4), die als einziger auf regulären Ausdrücken basierender Formalismus parametrisierte Ausdrücke und Quantoren unterstützen.

In den nachfolgenden Abschnitten (§ 6.2.1 bis § 6.2.4) werden die genannten Formalismen jeweils kurz vorgestellt, und es wird erläutert, mit welchen formalen Methoden ihre Semantik definiert wird und wie die Formalismen implementiert werden können. Anschließend folgt jeweils ein mehr oder weniger umfangreicher Vergleich mit Interaktionsausdrücken. Die Resultate dieser Vergleiche werden in § 6.2.5 zusammengefaßt.

6.2.1 Pfadausdrücke

Pfadausdrücke [Campbell74, Andler79, Campbell79, Campbell80] stellen einen ersten bedeutenden Versuch dar, die einfachen Synchronisationsmechanismen der Ebene 2 zu verlassen und Synchronisationsbedingungen mit Hilfe erweiterter regulärer Ausdrücke auf einer wesentlich abstrakteren und anwendungsnäheren Ebene zu spezifizieren.

6.2.1.1 Angebotene Operatoren

In ihrer Grundform [Campbell74] unterstützen Pfadausdrücke (in der Terminologie von Interaktionsausdrücken) die Operatoren für *sequentielle Komposition*, *Disjunktion* sowie sequentielle und parallele *Iteration* (vgl. Tab. 6.1). Als atomare Ausdrücke dienen Prozeduren einer parallelen imperativen Programmiersprache. Konjunktive Verknüpfungen im Sinne der *IAA-Synchronisation*[2] können durch Aneinanderreihen mehrerer Ausdrücke formuliert werden.

Bezeichnung	*Pfadausdruck*	*Interaktionsausdruck*
Sequentielle Komposition	$y; z$	$y - z$
Disjunktion	y, z	$y \bigcirc z$
Sequentielle Iteration	path y end	$\ominus y$
Parallele Iteration	$\{ y \}$	$\circledcirc y$
Synchronisation	$y \; z$	$y \boldsymbol{\circ} z$

Tabelle 6.1: Operatoren von Pfadausdrücken

Pfadausdrücke werden stets als Teil einer ADT- (Abstrakter Datentyp) bzw. Klassen-Definition formuliert und dann auf jedes Objekt dieses Typs separat angewandt. In der Terminologie von Interaktionsausdrücken bedeutet dies, daß jeder Ausdruck implizit mittels eines *parallelen Quantors* über alle Objekte des Typs quantifiziert wird.

[2] IAA steht für Interaktionsausdruck bzw. Interaktionsausdrücke.

Beispiel

Die folgende Lösung des elementaren Leser-Schreiber-Problems (zu jedem Zeitpunkt darf entweder ein Schreiber oder beliebig viele Leser auf ein Datenobjekt zugreifen [Courtois71]) stellt ein einfaches Beispiel eines Pfadausdrucks dar:

```
path { read }, write end
```

Transformiert man diesen Ausdruck gemäß Tab. 6.1 und berücksichtigt die oben erwähnte implizite Quantifizierung von Pfadausdrücken, so erhält man den entsprechenden Interaktionsausdruck:

$$\underset{p}{\circledcirc} \circlearrowleft (\circledcirc\ \text{read}(p) \circ \text{write}(p)).$$

6.2.1.2 Definition der Semantik

Ursprünglich wurde die Semantik von Pfadausdrücken nur verbal bzw. durch ihre Abbildung auf Semaphoroperationen (siehe unten) definiert [Campbell74, Campbell79]. Später kamen Vorschläge für eine präzisere formale Behandlung hinzu, wie z. B. die Abbildung (einer Teilklasse) von Pfadausdrücken auf Petrinetze [Lauer75] (vgl. auch § 6.5.1.8) oder eine denotationale und axiomatische Definition [Berzins77][3].

6.2.1.3 Implementierung des Formalismus

Eine gegebene Menge von Pfadausdrücken kann sehr einfach und direkt in Folgen von Semaphoroperationen `P` und `V` übersetzt werden, die als Pro- bzw. Epiloge der in den Ausdrücken vorkommenden Prozeduren ausgeführt werden [Campbell74, Campbell79]. Für das einfache Leser-Schreiber-Problem

```
path { read }, write end
```

erhält man z. B. (in einer an die Programmiersprache `CH` angelehnten Notation):

```
sem s = 1, t = 1;       // Zwei Semaphore mit Initialwert 1.
int c = 0;              // Ein Zähler mit Initialwert 0.

proc read() {
    PP(c, s, t);        // Prolog.
    ......              // Rumpf von read().
    VV(c, s, t);        // Epilog.
}

proc write() {
    P(t);               // Prolog.
    ......              // Rumpf von write().
    V(t);               // Epilog.
}
```

mit den häufig benötigten Hilfsoperationen:[4]

```
proc PP(int& c, sem s, sem t) {
    P(s);               // Hilfssemaphor s erwerben.
    c++;                // Zähler c inkrementieren.
    if (c == 1) P(t);   // Ggf. Semaphor t erwerben.
    V(s);               // Hilfssemaphor s freigeben.
}
```

[3] Zitiert nach [Freisleben87].
[4] Der Typ `int&` bezeichnet eine *Referenz* auf `int`, d. h. der Parameter `c` wird jeweils *by reference* übergeben (vgl. VAR-Parameter in Modula).

```
proc VV(int& c, sem s, sem t) {
    P(s);               // Hilfssemaphor s erwerben.
    c--;                // Zähler c dekrementieren.
    if (c == 0) V(t);   // Ggf. Semaphor t freigeben.
    V(s);               // Hilfssemaphor s freigeben.
}
```

Pfadausdrücke werden in den Programmier- bzw. Spezifikationssprachen *Path-Pascal* [Campbell79, Campbell80] und *COSY* [Lauer79] praktisch eingesetzt.

6.2.1.4 Vergleich mit Interaktionsausdrücken

Ausdrucksmächtigkeit

Die grundlegenden Konzepte von Pfadausdrücken, nämlich zum einen die saubere Trennung der Synchronisationsaspekte von den eigentlichen Prozeßbeschreibungen und zum anderen die Idee, zur Beschreibung zulässiger Ausführungsreihenfolgen erweiterte reguläre Ausdrücke zu verwenden, wurden bei der Entwicklung von Interaktionsausdrücken übernommen. Ersetzt man die Begriffe *Prozedur* durch *Aktivität*, *Prozeß* (im Sinne eines i. w. sequentiellen Programms) durch *Workflow* und *System nebenläufiger Prozesse* (bzw. paralleles Programm) durch *Workflow-Geflecht* (vgl. § 5.4.1), so könnte man Pfadausdrücke unmittelbar zur Synchronisierung nebenläufiger Workflows einsetzen.

Auf den ersten Blick scheint auch die Ausdrucksmächtigkeit von Pfadausdrücken durchaus mit der von Interaktionsausdrücken vergleichbar zu sein. Immerhin bieten sie so wichtige Konzepte wie parallele Iteration und Quantifizierung an, und im direkten Vergleich fehlen im wesentlichen nur die parallele Komposition und der Disjunktions-Quantor (vgl. Tab. 6.2; die anderen beiden Quantoren spielen auch in Interaktionsausdrücken nur eine untergeordnete Rolle). Für einen fairen Vergleich der beiden Formalismen ist jedoch zu berücksichtigen, daß bei der *Kombination* der von Pfadausdrücken angebotenen Operatoren zum Teil erhebliche Einschränkungen zu berücksichtigen sind [Campbell74]:

- Zum ersten darf in einer zusammengehörenden Menge von Pfadausdrücken jeder Prozedurname *nur einmal* auftreten. Durch die Einführung von Dummy-Prozeduren, die lediglich eine andere Prozedur aufrufen, kann man diese Einschränkung zwar prinzipiell umgehen, die resultierenden Formulierungen verlieren allerdings wesentlich an Kompaktheit, Eleganz und Verständlichkeit.
- Die sequentielle Iteration *muß* einerseits als äußerster Operator jedes Ausdrucks angewandt werden, andererseits darf sie *nicht* verschachtelt werden. Dies erklärt auch ihre zunächst etwas merkwürdig anmutende Notation mit den Worten `path` und `end`, die im Prinzip als Schlüsselworte zur Einbettung eines Pfadausdrucks in die umgebende ADT-Definition dienen.
- Die parallele Iteration darf ebenfalls nicht verschachtelt werden.

Formalismus	Sequ. Komp.	Sequ. Iter.	Par. Komp.	Par. Iter.	Disj.	Konj.	Sync.	Quant.
Pfadausdrücke	+	+	−	+	+	−	+	+
Interaktionsausdrücke	+	+	+	+	+	+	+	+

Legende	
+	Operator wird vollständig unterstützt
+	Operator wird eingeschränkt unterstützt
−	Operator wird nicht unterstützt

Tabelle 6.2: Vergleich von Pfad- und Interaktionsausdrücken

- Konjunktive Verknüpfungen sind nur auf äußerster Ebene, d. h. durch Aneinanderreihen von Ausdrücken, zulässig.
- Die implizite Quantifizierung über alle Objekte eines Typs ist einerseits (ähnlich wie die sequentielle Iteration) zwingend und kann andererseits weder verschachtelt noch irgendwie modifiziert werden. Für einfache Beispiele (wie etwa das Leser-Schreiber-Problem von oben) mag diese Form der Quantifizierung angemessen und ausreichend sein, zur Lösung komplexerer Probleme erweist sie sich jedoch als zu starr.

Beispiel

Die Summe dieser Einschränkungen führt zwangsläufig dazu, daß sich viele interessante Probleme, die mit Interaktionsausdrücken auf einfache und natürliche Art und Weise gelöst werden können, mit Pfadausdrücken nicht oder nur sehr umständlich lösen lassen. Als Beispiel betrachte man die von Campbell und Habermann [Campbell74] vorgeschlagene Lösung des Leser-Schreiber-Problems mit Schreiberpriorität (wenn sowohl Leser als auch Schreiber auf die Beendigung einer Schreiboperation warten, erhalten Schreiber anschließend „Vorfahrt“):

```
path readattempt end
path requestread, { requestwrite } end
path { openread; READ }, WRITE end
```

mit den Hilfsprozeduren:

```
proc readattempt() { requestread(); }
proc requestread() { openread(); }
proc openread() { }
proc requestwrite() { WRITE(); }

proc read() { readattempt(); READ(); }
proc write() { requestwrite(); }
```

Hierbei repräsentieren `READ()` und `WRITE()` die eigentlichen (internen) Lese- bzw. Schreiboperationen, während `read()` und `write()` die nach außen hin sichtbaren Schnittstellen darstellen.

Zum Vergleich betrachte man eine äquivalente Formulierung mit Interaktionsausdrücken, bei der im Prinzip die einfache Leser-Schreiber-Bedingung rw(p) aus § 6.2.1.1 mittels modularer Kombination um eine Priorisierungsklausel prio(p) erweitert wird (vgl. auch Abb. 6.3):

$$\text{rw}(p) \equiv \circlearrowleft \{ \circledcirc\, \text{READ}(p) \bigcirc \text{WRITE}(p) \},$$

$$\text{prio}(p) \equiv \circlearrowleft \{ \circledcirc\, [\text{wantwrite}(p) - \text{WRITE}(p)] \bigcirc \text{wantread}(p) \},$$

$$\underset{p}{\circledcirc} (\text{rw}(p) \bullet \text{prio}(p)).$$

Die exportierten Prozeduren `read()` und `write()` müssen hierfür wie folgt definiert werden:

```
proc read() { wantread(); READ(); }
proc write() { wantwrite(); WRITE(); }
```

wobei `wantread()` und `wantwrite()` leere Hilfsprozeduren darstellen.

Verglichen mit der Pfadausdruck-Lösung von oben, die kaum einfacher oder besser verständlich ist als eine unmittelbar auf Semaphoren basierende Formulierung, ist die IAA-Lösung doch um einiges kompakter und übersichtlicher und benötigt darüber hinaus nur zwei einfache Hilfsprozeduren.

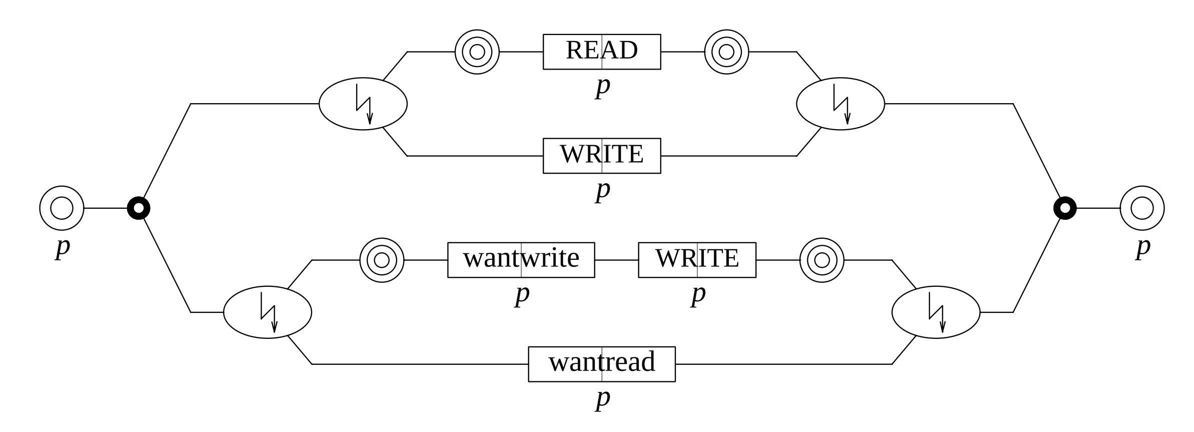


Abbildung 6.3: Leser-Schreiber-Problem mit Schreiberpriorität

Orthogonalität

Abgesehen von Kriterien wie Einfachheit und Eleganz, deren Bewertung zwangsläufig subjektiven Einflüssen unterliegt, behindert ein Formalismus, dessen Operatoren nicht orthogonal kombiniert werden können, in erheblichem Maße die *modulare Kombination* von Ausdrücken, d. h. das nachträgliche Zusammenfügen oder Ineinandereinsetzen von unabhängig entwickelten Teilausdrücken, also insbesondere auch das wichtige Prinzip der Abstraktion (vgl. § 2.8.2.1).

Campbell und Habermann rechtfertigen ihre, auf den ersten Blick recht willkürlich erscheinenden Einschränkungen einerseits damit, daß sie für die von ihnen betrachteten Beispiele nicht sehr gravierend seien; zum anderen − und das ist der entscheidende Grund − ermöglichen diese Einschränkungen die oben erwähnte direkte und effiziente Abbildung auf Semaphoroperationen (oder evtl. auch andere Synchronisationsmechanismen der Ebene 2).

Beim Entwurf von Interaktionsausdrücken spielten derartige Überlegungen nur eine sehr untergeordnete Rolle. So ist ihre vollständige Implementierung in der Tat vergleichsweise aufwendig und natürlich bei weitem nicht so effizient wie „eine Handvoll“ Semaphoroperationen. Allerdings muß man berücksichtigen, daß in Anwendungsbereichen wie Workflow-Management − im Gegensatz beispielsweise zu kritischen Betriebssystemkomponenten − Effizienz sicherlich nicht in Einheiten von Hardware-Instruktionen gemessen werden darf. Auf der anderen Seite standen bei der Entwicklung von Interaktionsausdrücken Konzepte wie *Ausdrucksmächtigkeit*, *Orthogonalität*, d. h. beliebige Kombinierbarkeit der angebotenen Operatoren, und *Abstraktion*, d. h. Zusammenfassen und Verbergen komplexer Teilausdrücke, eindeutig im Vordergrund.

6.2.1.5 Erweiterungen und Variationen

Im Laufe der Zeit wurden verschiedene Erweiterungen und Variationen von Pfadausdrücken vorgeschlagen, insbesondere *predicate path expressions* [Andler79] und *open path expressions* [Campbell79, Campbell80]. Bei letzteren gelang es zwar − unter Beibehaltung der einfachen und effizienten Implementierbarkeit −, einen vollständig orthogonalen Formalismus bereitzustellen, aber durch den Verzicht auf explizite Iterationen bleibt die Ausdrucksmächtigkeit nach wie vor stark eingeschränkt. Beispielsweise läßt sich das einfache Leser-Schreiber-Problem mit *open path expressions* nicht mehr lösen, da parallele Iteration nur noch implizit auf äußerster Ebene (ähnlich wie ursprünglich die sequentielle Iteration) unterstützt wird.

6.2.2 Synchronisierungsausdrücke

Synchronisierungsausdrücke [Govindarajan91, Guo95, Guo96] stellen eine moderne Variante von Pfadausdrücken dar, bei denen einerseits versucht wurde, unnötige und für Anwender meist uneinsichtige Einschränkungen zu vermeiden, und andererseits großer Wert auf eine präzise formale Semantik gelegt wurde.

6.2.2.1 Angebotene Operatoren

Das Spektrum der angebotenen Operatoren umfaßt, wiederum in der Terminologie dieser Arbeit, *sequentielle* und *parallele Komposition*, die Booleschen Operationen *Disjunktion* und *Konjunktion*, sowie *sequentielle Iteration*. Die parallele Iteration war ursprünglich offenbar vorgesehen [Govindarajan90, Govindarajan91], wurde aber in der endgültigen Version [Guo95, Guo96, Yu96] wieder verworfen, so daß Synchronisierungsausdrücke eine Teilmenge quasi-regulärer Ausdrücke darstellen (vgl. § 4.7.2.1).

Die Autoren betonen, daß die atomaren Bestandteile von Synchronisierungsausdrücken keine Prozeduren (wie bei Pfadausdrücken), sondern *einzelne Anweisungen* sind, die mit sogenannten *Statement tags* (vergleichbar, aber nicht identisch zu Goto labels) gekennzeichnet werden. Dies hat zwar den Vorteil, daß Synchronisationsbedingungen mit feinerer Granularität formuliert werden können, führt aber auf der anderen Seite fast zwangsläufig dazu, daß Synchronisations- und normaler Anwendungscode vermischt werden.

Die angebotenen Operatoren sind *fast* orthogonal; eine nicht zu vernachlässigende Einschränkung besteht jedoch darin, daß die Operanden einer parallelen Komposition disjunkte Alphabete aufweisen müssen. Diese Restriktion, die vermutlich durch Implementierungsaspekte motiviert ist, verbietet insbesondere die Definition paralleler Multiplikatoren (Mehrfach-Verzweigungen) als

$$\overset{n}{\circledcirc}\, x = x \circledcirc \ldots \circledcirc x \; (n\text{-mal } x),$$

da hier die Operanden der parallelen Komposition sogar identisch sind.

6.2.2.2 Definition der Semantik

Zur Definition der Semantik von Synchronisierungsausdrücken wird, ähnlich wie in § 3.3, ein sprachtheoretischer Ansatz verwendet [Guo95, Guo96]. Die Idee, zeitlich ausgedehnte Aktivitäten als Folge zweier punktueller Aktionen zu betrachten, findet sich dort in gleicher Weise. Allerdings werden, der traditionellen Sichtweise folgend, nur *vollständige* Worte betrachtet.

Darüber hinaus werden die resultierenden *synchronization languages* mit allgemeineren *start-termination languages* verglichen und einige, hier nicht weiter interessierende Abschlußeigenschaften vorgestellt.

6.2.2.3 Implementierung des Formalismus

Da die Hinzunahme von paralleler Komposition und Konjunktion formal keine Erweiterung regulärer Ausdrücke darstellt (vgl. § 3.5.2), lassen sich Synchronisierungsausdrücke direkt auf deterministische oder nichtdeterministische endliche Automaten abbilden und auf diese Weise auch effizient implementieren. Hierin liegt vermutlich auch der Grund für den Verzicht auf den parallelen Iterationsoperator, dessen Hinzunahme die Ausdrucksmächtigkeit und den Implementierungsaufwand erheblich erhöhen würde.

Synchronisierungsausdrücke werden in der Programmiersprache *ParC* [Govindarajan90, Govindarajan91] praktisch eingesetzt.

6.2.2.4 Vergleich mit Interaktionsausdrücken

Im Vergleich zu Interaktionsausdrücken fehlt bei Synchronisierungsausdrücken vor allem, wie bereits erwähnt, die *parallele Iteration* sowie das Konzept der *Quantoren*. Außerdem wird die Konjunktion nur in ihrer oftmals unhandlichen *strikten* Form angeboten, d. h. es fehlt der für die Praxis wesentliche *Synchronisationsoperator* (vgl. Tab. 6.4).

Formalismus	Sequ. Komp.	Sequ. Iter.	Par. Komp.	Par. Iter.	Disj.	Konj.	Sync.	Quant.
Synchronisierungsausdr.	+	+	+	−	+	+	−	−
Interaktionsausdrücke	+	+	+	+	+	+	+	+

Tabelle 6.4: Vergleich von Synchronisierungs- und Interaktionsausdrücken

Die Unterscheidung zwischen einzelnen Anweisungen und ganzen Prozeduren als Granulat der Synchronisierung mag auf den ersten Blick nebensächlich erscheinen. Ein wesentlicher Unterschied besteht jedoch darin, daß *Anweisungen* zum Zwecke der Synchronisation mit einem Statement tag markiert werden müssen, was u. U. einen nachträglichen Eingriff in den Programmcode erfordert, während *Prozeduren* in einem Ausdruck einfach durch ihren Namen referenziert werden können. Auf die Synchronisation von Workflows übertragen, bedeutet dies, daß man im ersten Fall die Beschreibungen der zu synchronisierenden Workflows um explizite Markierungen erweitern müßte, während man sie im zweiten Fall unverändert übernehmen kann. Aus diesem Grund wird in dieser Arbeit gerade die entgegengesetzte Meinung von Guo et al. vertreten, indem die Synchronisierung auf der Ebene von Prozeduren bzw. Aktivitäten bevorzugt wird.

6.2.3 Ereignis- und Flußausdrücke

Ereignis- und *Flußausdrücke* [Riddle73, Shaw78, Shaw80b] stammen zwar von unterschiedlichen Autoren, sind sich aber konzeptuell so ähnlich, daß es gerechtfertigt scheint, sie im folgenden gemeinsam zu behandeln.

6.2.3.1 Angebotene Operatoren

Die „Schnittmenge“ der beiden Formalismen besteht aus den Operatoren für *sequentielle* und *parallele Komposition* und *Iteration* sowie *Disjunktion*. Im Vergleich zu Interaktionsausdrücken fehlen also nur die beiden Formen der Konjunktion (Konjunktion und Synchronisation) sowie das Konzept der Quantoren. Stattdessen gibt es in beiden Formalismen zusätzliche Konstrukte zur Synchronisation paralleler Zweige, die das Konzept der Konjunktion entbehrlich machen.

In *Ereignisausdrücke* können spezielle *Synchronisationssymbole* (dort mit $@$, $\overline{@}$, $@_1$, $\overline{@}_1$ usw. bezeichnet) eingebettet werden, die bei der Verschränkung von Worten wie folgt behandelt werden: Paare $@_i\overline{@}_i$ und $\overline{@}_i@_i$ von aufeinanderfolgenden komplementären Synchronisationssymbolen werden entfernt, und anschließend werden nur Worte akzeptiert, die keine Synchronisationssymbole mehr enthalten. Auf diese Weise kann die Menge der möglichen Verschränkungen von Worten mehr oder weniger stark eingeschränkt werden.

Im Gegensatz zu diesem deklarativen Ansatz wird in *Flußausdrücken* das „gute alte“ Semaphorkonzept aufgegriffen und in Form von Signal- und Wait-Operationen σ_i und ω_i angeboten, die ähnlich wie die obigen Synchronisationssymbole an beliebigen Stellen eines Ausdrucks stehen können und wie folgt interpretiert werden: Ein Wort wird nur akzeptiert, wenn jedem Wait-Symbol ω_i mindestens ein zugehöriges Signal-Symbol σ_i vorausgeht. Aus Gründen der Bequemlichkeit gibt es außerdem

spezielle Klammersymbole $[_i$ und $]_i$, mit denen direkt − ohne explizite Verwendung von Semaphoren − kritische Regionen markiert werden können.

Neben diesen Synchronisationsmechanismen gibt es in Flußausdrücken noch einen sogenannten *Cyclic-Operator*, mit dem − im Gegensatz zur gewöhnlichen sequentiellen Iteration − explizit *unendliche Iterationen* beschrieben werden können. Da dieser Operator aber sinnvollerweise nur auf äußerster Ebene angewandt wird und dort nahezu äquivalent zum einfachen Iterationsoperator ist, ist sein wirklicher praktischer Nutzen sehr fraglich. Seine Verwendung hat daher eher dokumentierenden Charakter.

6.2.3.2 Definition der Semantik

Die Semantik von Ereignis- und Flußausdrücken wird ebenfalls durch einen traditionellen sprachtheoretischen Ansatz definiert [Shaw78]. Da der oben erwähnte Cyclic-Operator jedoch auch auf Teilausdrücke angewandt werden kann, muß durchweg mit potentiell unendlichen Worten gearbeitet werden.

Die speziellen Synchronisationssymbole $@_i$, $\overline{@}_i$, σ_i und ω_i werden zunächst wie normale Symbole behandelt. Anschließend werden aus der resultierenden Sprache diejenigen Worte entfernt, die die oben erwähnten Regeln verletzen. Die Klammersymbole $[_i$ und $]_i$ werden entsprechend behandelt oder durch äquivalente Signal- und Wait-Symbole ersetzt.

6.2.3.3 Implementierung des Formalismus

Ereignis- und Flußausdrücke wurden primär als Software-Spezifikations- und Dokumentationssprachen entwickelt [Riddle73, Riddle79ab, Shaw78, Shaw80b], für die die Existenz einer realen Implementierung nur eine untergeordnete Rolle spielte.

Dennoch nennt Shaw in [Shaw80a] den in § 4.4.1 vorgestellten Algorithmus zur Lösung des Wortproblems für Flußausdrücke, allerdings ohne Verwendung von Klammer-, Signal- und Wait-Symbolen (d. h. einer echten Teilmenge von Interaktionsausdrücken). Obwohl der Algorithmus außergewöhnlich kompakt und elegant ist, scheitert seine praktische Verwendbarkeit, wie in § 4.4.4 erläutert, an seiner nicht beherrschbaren Komplexität.

6.2.3.4 Vergleich mit Interaktionsausdrücken

Ausdrucksmächtigkeit

Ereignis- und Flußausdrücke sind die einzigen bekannten Ausdrucksformalismen, bei denen das Konzept der *Nebenläufigkeit* konsequent und vollständig unterstützt wird, indem die parallele Komposition/Iteration einfach als nebenläufiges Analogon zur sequentiellen Komposition/Iteration betrachtet wird [Shaw80b].

Abgesehen von Quantoren, besteht der wesentliche Unterschied zwischen Interaktionsausdrücken und Ereignis-/Flußausdrücken daher nur darin, daß erstere *konjunktiv verknüpft*[5] werden können, während letztere stattdessen zusätzliche *Synchronisationsmechanismen* anbieten (vgl. Tab. 6.5). Laut

Formalismus	Sequ. Komp.	Sequ. Iter.	Par. Komp.	Par. Iter.	Disj.	Konj.	Sync.	Quant.	Synchr.-Symbole
Ereignisausdr.	+	+	+	+	+	−	−	−	+
Flußausdrücke	+	+	+	+	+	−	−	−	+
Interaktionsausdr.	+	+	+	+	+	+	+	+	−

Tabelle 6.5: Vergleich von Ereignis-, Fluß- und Interaktionsausdrücken

[5] Da eine Synchronisation intuitiv ebenfalls eine *konjunktive Verknüpfung* von Ausdrücken darstellt (vgl. § 3.3.1.10), wird dieser Begriff im folgenden als Oberbegriff von Konjunktion und Synchronisation verwendet.

[Shaw80b] sind diese Mechanismen in beiden Fällen so mächtig, daß sich jede rekursiv aufzählbare Menge sowohl mit Ereignis- als auch mit Flußausdrücken beschreiben läßt, d. h. daß beide Formalismen *berechnungsuniversell* sind. Obwohl dies für Interaktionsausdrücke vermutlich nicht zutrifft (vgl. § 3.5.4), gab es bei ihrer Entwicklung dennoch eine Reihe von Gründen, auf derartige Mechanismen zu verzichten und stattdessen konjunktive Verknüpfungen zu integrieren:

- Von einem ästhetischen Standpunkt aus betrachtet, wirkt zumindest das Semaphorkonzept bei *Flußausdrücken*, das ja prozeduraler oder imperativer Natur ist, in einem ansonsten deklarativen Ausdrucks-Formalismus fehl am Platze. Wie in § 6.1.1 erwähnt, wurden Synchronisationsmechanismen der Ebene 3, der die hier diskutierten Ausdrucks-Formalismen angehören, ja gerade entwickelt, um Anwendern die „Niederungen“ der Ebene 2 (der Semaphore angehören) zu ersparen.
- Das bei *Ereignisausdrücken* verwendete Konzept ist zwar eher deklarativer Natur, führt aber dennoch oft zu Formulierungen, die tendenziell schwerer verständlich sind als solche, die stattdessen konjunktive Verknüpfungen verwenden (vgl. das folgende Beispiel).
- Bei beiden Konzepten muß ein Ausdruck, für den die Menge seiner zulässigen Ausführungsreihenfolgen weiter eingeschränkt werden soll, um zusätzliche Synchronisationssymbole erweitert, d. h. unter Umständen *nachträglich verändert* werden. Bei einer konjunktiven Verknüpfung hingegen, wie sie von *Interaktionsausdrücken* unterstützt wird, kann der ursprüngliche Ausdruck unverändert übernommen werden. Somit ist das Konzept der konjunktiven Verknüpfung das einzige, das das Prinzip der *modularen Kombination* von Ausdrücken konsequent unterstützt.
- Die Semantik der Konjunktion und Synchronisation ist wesentlich leichter zu verstehen als die der obigen Synchronisationsmechanismen. Die Tatsache, daß die entsprechenden Operatoren darüber hinaus auch einfacher zu implementieren sind, ist ein willkommener Nebeneffekt, war für die konzeptuellen Überlegungen jedoch nicht ausschlaggebend.

Abgesehen von diesen Gründen, wurde bei der Entwicklung von Interaktionsausdrücken nie das Ziel verfolgt, einen berechnungsuniversellen Formalismus zu entwickeln.

Beispiel

Das folgende einfache Beispiel (entnommen aus [Shaw80b]) zum Vergleich von Ereignis-, Fluß- und Interaktionsausdrücken demonstriert insbesondere das Prinzip der modularen Kombination von Ausdrücken.

Gegeben sei ein Erzeuger-Prozeß, der abwechselnd die Aktivitäten produce (zum Erzeugen irgendeines Objektes) und put (zum Ablegen eines erzeugten Objekts in einem Zwischenpuffer) ausführt. Gegeben sei weiterhin ein Verbraucher-Prozeß, der abwechselnd die Aktivitäten get und consume ausführt, um jeweils ein Objekt aus dem Zwischenpuffer zu holen und anschließend irgendwie zu verbrauchen. Die zulässigen Ausführungsreihenfolgen dieser beiden Prozesse können in IAA-Notation durch die Ausdrücke

$\ominus$ (produce – put) bzw. $\ominus$ (get – consume)

beschrieben werden. Sofern der Zwischenpuffer jeweils nur ein Objekt aufnehmen kann, müssen auch die Aktivitäten put und get alternierend ausgeführt werden:

$\ominus$ (put – get).

Zur Beschreibung des Gesamtsystems mit Hilfe von *Interaktionsausdrücken* werden diese drei Teilausdrücke einfach „intuitiv konjunktiv“ mittels Synchronisation verknüpft:

$\ominus$ (produce – put) $\boldsymbol{\circ}$ $\ominus$ (get – consume) $\boldsymbol{\circ}$ $\ominus$ (put – get).

Bei der Verwendung von *Ereignisausdrücken* müssen die beiden ersten Teilausdrücke zunächst um zusätzliche Symbole erweitert werden, die die zu synchronisierenden Aktivitäten put und get umge-

ben:

$$\ominus\left(\text{produce} - @_1 - \text{put} - @_2\right) \quad \text{bzw.} \quad \ominus\left(@_3 - \text{get} - @_4 - \text{consume}\right).$$

Die eigentliche Synchronisationsbedingung verwendet dann gerade die komplementären Symbole:

$$\ominus\left(\overline{@}_1 - \overline{@}_2 - \overline{@}_3 - \overline{@}_4\right),$$

und mittels *paralleler Komposition* erhält man den Gesamtausdruck

$$\ominus\left(\text{produce} - @_1 - \text{put} - @_2\right) \circledcirc \ominus\left(@_3 - \text{get} - @_4 - \text{consume}\right) \circledcirc \ominus\left(\overline{@}_1 - \overline{@}_2 - \overline{@}_3 - \overline{@}_4\right).$$

Bei Verwendung von *Flußausdrücken* geht man im Prinzip sehr ähnlich vor, allerdings synchronisieren sich die entsprechenden Signal- und Wait-Symbole selbst, so daß der dritte Teilausdruck entfällt:

$$\sigma_1 - \left\{ \ominus\left(\text{produce} - \omega_1 - \text{put} - \sigma_2\right) \circledcirc \ominus\left(\omega_2 - \text{get} - \sigma_1 - \text{consume}\right) \right\}.$$

Man beachte jedoch die Notwendigkeit des vorangestellten σ_1 zur Initialisierung des Semaphors 1. Diese Lösung unterscheidet sich nur noch syntaktisch von einer gewöhnlichen Semaphorlösung des Erzeuger-Verbraucher-Problems!

Die Lösung mit *Ereignisausdrücken* besitzt immerhin den Vorteil, daß die eigentliche Synchronisationsbedingung separat im dritten Teilausdruck der parallelen Komposition formuliert werden kann, was sich positiv auf ihre Änderungsfreundlichkeit auswirkt. Wollte man beispielsweise den einelementigen durch einen unbegrenzten Puffer ersetzen, so müßte man nur diesen dritten Teilausdruck durch den Ausdruck

$$⊚\left(\overline{@}_1 - \overline{@}_2 - \overline{@}_3 - \overline{@}_4\right)$$

ersetzen, während man bei der Lösung mit *Flußausdrücken* den kompletten Ausdruck umformulieren müßte:

$$\ominus\left(\text{produce} - \text{put} - \sigma_2\right) \circledcirc \ominus\left(\omega_2 - \text{get} - \text{consume}\right).$$

Im Vergleich zur Formulierung mit *Interaktionsausdrücken*, bei der man die ursprünglichen Beschreibungen des Erzeuger- und Verbraucherprozesses einfach konjunktiv mit der Synchronisationsbedingung des Puffers verknüpft, leiden beide anderen Lösungen jedoch daran, daß die ursprünglichen Prozeßbeschreibungen modifiziert werden müssen. Außerdem ist es schwierig, wenn nicht unmöglich, mit Ereignis- oder Flußausdrücken *allgemeine Integritätsbedingungen* für Aktivitäten wie put und get zu formulieren, deren Gültigkeit *unabhängig* von der konkreten Verwendung dieser Aktivitäten ist (vgl. § 5.4.3).

Implementierung

Die Existenz einer real einsetzbaren, da effizienten Implementierung ist ein eindeutiger Vorzug von Interaktionsausdrücken gegenüber Ereignis- und Flußausdrücken. Auf diese Weise ist es möglich, Ausdrücke nicht nur zur Spezifikation eines Systems, sondern auch für seine Implementierung zu verwenden.

Wie bereits in § 4.8.1.1 erläutert, besteht der wesentliche Unterschied zwischen Shaws Algorithmus und der in § 4.6 vorgestellten IAA-Implementierung darin, daß letztere sowohl den gegebenen Ausdruck als auch das konkret vorliegende Wort (bzw. die jeweils nächste Aktion dieses Wortes) dazu verwenden kann, den oft tatsächlich exponentiell großen Suchraum an Möglichkeiten so weit wie möglich einzuschränken. Kombiniert mit zusätzlichen Zustandsoptimierungen (§ 4.8.1.4), die darauf zielen, möglichst jede Redundanz innerhalb eines Zustandsbaums zu vermeiden, führt dies dazu, daß das Aktionsproblem für sehr viele Ausdrücke in konstanter oder linearer Zeit gelöst werden kann, sofern der Ausdruck keine parallelen Quantoren enthält. Bei einer zusätzlichen (vollständigen und homogenen) „Allquantifizierung“ auf äußerster Ebene ergibt sich dann schlimmstenfalls lineare bzw. quadratische Zeit (vgl. die Beweise in § 4.7.3 und 4.7.4).

Ein weiterer wichtiger Unterschied zwischen Shaws Algorithmus und der IAA-Implementierung besteht darin, daß ersterer in der publizierten Form nicht in der Lage ist, das Teilwort- oder *Aktionsproblem* zu lösen, obwohl diese Fragestellung auch in den von Shaw skizzierten Anwendungsbereichen von großer Bedeutung ist. Diese Tatsache untermauert die oben erwähnte Aussage, daß Ereignis- und Flußausdrücke nur zur Spezifikation oder zur Dokumentation, nicht jedoch zur realen Implementierung von Software-Systemen eingesetzt werden.

6.2.4 CoCoA-Ausführungsregeln

CoCoA (Coordinated Cooperation Agendas) [Faase96] ist eine Spezifikationssprache für kooperative Szenarien − wie z. B. die gemeinsame Erstellung eines größeren Dokuments durch mehrere Autoren −, die im Rahmen des ESPRIT-Projekts *TransCoop* an der Universität Twente entwickelt wurde. Neben umfangreichen Beschreibungsmitteln zur Spezifikation kooperierender Aktivitäten, bietet die Sprache sogenannte *Ausführungsregeln* an, die eine Teilmenge von Interaktionsausdrücken darstellen.

6.2.4.1 Angebotene Operatoren

Eine Ausführungsregel ist ein *deterministischer regulärer Ausdruck*, d. h. ein Ausdruck, der die Operatoren *Disjunktion* und *sequentielle Komposition* und *Iteration* enthalten kann und *direkt* (ohne Umweg über einen nichtdeterministischen Automaten) in einen deterministischen endlichen Automaten transformiert werden kann. Ausdrücke wie z. B. $(a - b) \bigcirc (a - c)$ oder $\ominus\, a - a - b$, bei deren Abarbeitung u. U. mehrere Alternativen gleichzeitig verfolgt werden müssen, sind nicht zulässig.[6]

Aus Gründen der Bequemlichkeit wird die *beliebige Reihenfolge* (n Aktivitäten $A_1, \ldots, A_n$ dürfen sequentiell in beliebiger Reihenfolge ausgeführt werden) als zusätzlicher Operator angeboten, wodurch die prinzipielle Ausdrucksmächtigkeit jedoch nicht erhöht wird.

Ausführungsregeln können *Parameter* enthalten, über die auf äußerster Ebene mittels eines `forall`-Quantors quantifiziert werden kann, dessen Semantik dem *Synchronisationsquantor* von Interaktionsausdrücken entspricht. Ebenso wird eine *Menge* von Ausführungsregeln implizit nach *Synchronisationssemantik* verknüpft.

Der spezielle Parameterwert `'_'` kann verwendet werden um anzuzeigen, daß für diesen Parameter ein *beliebiger Wert* zulässig ist. Dies entspricht einer eingeschränkten Variante des *Disjunktionsquantors*, bei dem der Rumpf aus einer einzigen Aktion besteht. Wenn ein Wert zwar beliebig, aber für mehrere Aktionen *gleich* sein soll, so kann dies indirekt mittels eines zusätzlichen `forall`-Ausdrucks formuliert werden. Beispielsweise entspricht dem Interaktionsausdruck

$$\underset{f}{\circledcirc}\ \ominus\ \underset{p}{\bigcirc}\,[\mathrm{get}(p,\ f) - \mathrm{put}(p,\ f)] \qquad \text{(vgl. § 5.3.2.2)}$$

die Menge der beiden CoCoA-Regeln:

```
forall f : Fork
order ( get(_, f) ; put(_, f) ) *

forall f : Fork, p : Phil
order ( get(p, f) ; put(p, f) ) *
```

6.2.4.2 Definition der Semantik

Die Semantik einer einfachen Ausführungsregel (ohne Quantifizierung) wird im Prinzip durch die dem Ausdruck entsprechende reguläre Menge definiert. Ebenso wie bei Interaktionsausdrücken, interessiert man sich jedoch primär für die partiellen und weniger für die vollständigen Worte eines Ausdrucks. Da die in § 2.5 und § 3.2.3 genannten Sonderfälle (Sackgassen und endlose Wege) aufgrund

[6] Allerdings wird kein syntaktisches Kriterium zur Aussonderung solcher Ausdrücke genannt.

der eingeschränkten Ausdrucksmächtigkeit nicht auftreten, kann die Menge $\Psi(x)$ jedoch mittels Präfixbildung aus der Menge $\Phi(x)$ abgeleitet werden und muß nicht separat definiert werden.

Die Semantik einer *Menge* von Regeln wird, ähnlich wie die IAA-Synchronisation, mittels Projektion und Durchschnittsbildung definiert, wobei quantifizierte Regeln im Prinzip als Mengen von unendlich vielen einzelnen Regeln betrachtet werden.

6.2.4.3 Implementierung des Formalismus

Die gesamte Spezifikationssprache CoCoA, einschließlich der hier vorgestellten Ausführungsregeln, wird durch eine Abbildung auf die formale Spezifikationssprache LOTOS/TM [Even95] implementiert. LOTOS/TM wiederum basiert auf der Prozeßalgebra-Sprache LOTOS (Language of Temporal Ordering Specification) [ISO87, Bolognesi87, Vissers91] sowie auf der Datenbankspezifikationssprache TM [Balsters91, Balsters93, Bal95].[7] Die oben erwähnte Einschränkung auf *deterministische* reguläre Ausdrücke liegt darin begründet, daß sich nur solche Ausdrücke unmittelbar auf LOTOS/TM abbilden lassen.

6.2.4.4 Vergleich mit Interaktionsausdrücken

Verglichen mit den übrigen hier diskutierten Ausdrucks-Formalismen, sind die Ausführungsregeln von CoCoA die einzigen, die das Konzept der *parametrisierten Ausdrücke* und *Quantoren* explizit und relativ allgemein unterstützen. Im Vergleich zu Interaktionsausdrücken fehlt jedoch ein expliziter *Disjunktionsquantor* sowie die Möglichkeit, Quantoren auch auf *Teilausdrücke* anwenden zu können (vgl. Tab. 6.6). Der spezielle Parameterwert `'_'` ist hierfür nur ein notdürftiger Ersatz, wie das obige Beispiel verdeutlicht, bei dem nicht unmittelbar klar ist, daß nur die beiden `forall`-Ausdrücke *zusammen* die gewünschte Integritätsbedingung spezifizieren.

Formalismus	Sequ. Komp.	Sequ. Iter.	Par. Komp.	Par. Iter.	Disj.	Konj.	Sync.	Quant.
CoCoA-Ausführungsregeln	+	+	−	−	+	−	+	+
Interaktionsausdrücke	+	+	+	+	+	+	+	+

Tabelle 6.6: Vergleich von CoCoA-Ausführungsregeln und Interaktionsausdrücken

Abgesehen davon ist die Ausdrucksmächtigkeit der Regeln stark eingeschränkt, da nicht einmal *beliebige* reguläre Ausdrücke unterstützt werden. Theoretisch betrachtet, stellt die Forderung nach *deterministischen* regulären Ausdrücken zwar keine echte Einschränkung der Ausdrucksmächtigkeit dar, da deterministische und nichtdeterministische endliche Automaten bzw. Ausdrücke bekanntlich gleichmächtig sind; vom praktischen Standpunkt aus betrachtet, kann diese Einschränkung jedoch gravierend sein. Man denke in diesem Zusammenhang nur wieder an das bereits mehrfach erwähnte Prinzip der *modularen Kombination* von Ausdrücken!

6.2.5 Zusammenfassung

6.2.5.1 Ausdrucksmächtigkeit

Tabelle 6.7 faßt die vergleichenden Tabellen der vorangegangenen Abschnitte zusammen und ermöglicht so einen Gesamtüberblick über die Ausdrucksmächtigkeit der betrachteten Formalismen im Vergleich zu Interaktionsausdrücken. Wie zuvor wird für jeden Operator angegeben, ob er von einem Formalismus explizit und vollständig (+), nur implizit oder mit gewissen Einschränkungen (+) oder über-

[7] Alle Literaturangaben dieses Abschnitts stammen aus [Faase96].

Formalismus	Sequ. Komp.	Sequ. Iter.	Par. Komp.	Par. Iter.	Disj.	Konj.	Sync.	Quant.
Reguläre Ausdrücke	+	+	−	−	+	−	−	−
Pfadausdrücke	+	+	−	+	+	−	+	+
Synchronisierungsausdr.	+	+	+	−	+	+	−	−
Ereignisausdrücke	+	+	+	+	+	−	−	−
Flußausdrücke	+	+	+	+	+	−	−	−
CoCoA-Ausführungsregeln	+	+	−	−	+	−	+	+
Interaktionsausdrücke	+	+	+	+	+	+	+	+

Abbildung 6.7: Vergleich verschiedener Ausdrucksformalismen

haupt nicht unterstützt wird (−). Abgesehen von den speziellen Synchronisationsmechanismen von Ereignis- und Flußausdrücken (vgl. § 6.2.3.1), stellen die angegebenen Operatoren die Vereinigungsmenge aller betrachteten Formalismen dar. Abbildung 6.8 zeigt schematisch, wie man, ausgehend von regulären Ausdrücken als Basisformalismus, durch sukzessive Erweiterungen auf zwei komplementären Wegen zu Interaktionsausdrücken gelangen kann.

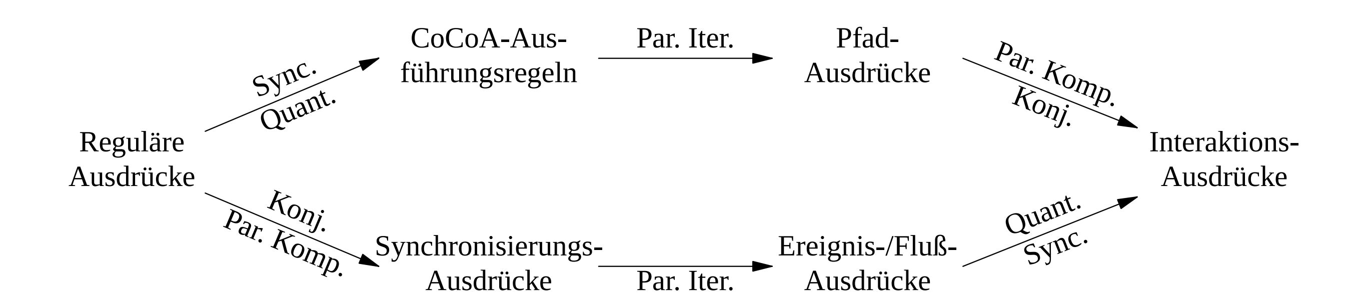


Abbildung 6.8: Zwei komplementäre Wege von regulären zu Interaktionsausdrücken

6.2.5.2 Vollständigkeit

Beide Darstellungen verdeutlichen, daß sämtliche bisher in der Literatur vorgeschlagenen Ausdrucksformalismen im Vergleich zu Interaktionsausdrücken *unvollständig* sind. Abgesehen von den speziellen Synchronisationsmechanismen von Ereignis- und Flußausdrücken, auf die beim Entwurf von Interaktionsausdrücken bewußt verzichtet wurde (vgl. § 6.2.3.4), stellen Interaktionsausdrücke daher eine echte *Obermenge* aller vergleichbaren Ansätze dar. Die Unvollständigkeit der anderen Ansätze erscheint umso merkwürdiger, wenn man bedenkt, daß oftmals von zwei logisch zusammengehörenden Operatoren nur einer unterstützt wird:

- Synchronisierungsausdrücke unterstützen zwar parallele *Komposition*, nicht jedoch den zugehörigen Hüllenoperator, die parallele *Iteration*.
- Ereignis- und Flußausdrücke unterstützen zwar *Disjunktion*, nicht jedoch den dualen Booleschen Operator, die *Konjunktion*.
- CoCoA-Ausführungsregeln unterstützen zwar *Allquantoren* (`forall`), nicht jedoch die dualen *Existenz-Quantoren* (`exists`).

Vor diesem Hintergrund wurde versucht, Interaktionsausdrücke als in sich *abgeschlossenen* und *vollständigen* Formalismus zu entwerfen:

- Es gibt zwei grundlegende Formen der *Komposition*: sequentielle und parallele Komposition.
- Es gibt zwei zugehörige *Hüllenoperatoren*: sequentielle und parallele Iteration.
- Es gibt zwei duale *Boolesche Operatoren*: Disjunktion und Konjunktion.
- Zu jedem kommutativen binären Operator gibt es einen zugehörigen *Quantor*, wobei All- und Existenzquantoren (d. h. parallele und Disjunktions-Quantoren) wichtige Spezialfälle darstellen.

Aus anderen Blickwinkeln betrachtet, ergeben sich weitere Dualitäten:

- Eine Disjunktion entspricht einer Verzweigung, bei der *genau ein* Zweig beschritten wird, während bei den übrigen Verzweigungsoperatoren (Konjunktion, Synchronisation, parallele Komposition) *beide* bzw. *alle* Zweige beschritten werden.
- Bei einer Konjunktion werden die Zweige *strikt synchronisiert*, während sie bei einer parallelen Komposition *vollkommen unabhängig* durchlaufen werden. Die Synchronisation stellt eine häufig benötigte *Zwischenform* dar.

6.2.5.3 Orthogonalität

Neben dem Problem der Unvollständigkeit leiden viele verwandte Ansätze an mangelnder *Orthogonalität*. Die negativen Konsequenzen hieraus, insbesondere im Blick auf die *modulare Kombination* von Ausdrücken, wurden bereits mehrfach erwähnt.

Aus diesem Grund wurden Interaktionsausdrücke *vollständig orthogonal* entworfen, d. h. an jeder Stelle eines Ausdrucks, an der ein atomarer Ausdruck (d. h. eine Aktion) stehen darf, darf auch ein *beliebig komplexer* Teilausdruck stehen (vgl. § 2.8.2.1). Insbesondere ist die Anwendbarkeit von konjunktiven Operatoren und Quantoren nicht − wie bei fast allen anderen Ansätzen − auf die oberste Ebene von Ausdrücken beschränkt.

6.2.5.4 Resümee

Sicherlich hätte man die Kernteile dieser Arbeit, d. h. die formale und operationale Semantik sowie die Implementierung von Interaktionsausdrücken, wesentlich vereinfachen und reduzieren können, wenn man der immer wieder aufgetretenen Versuchung erlegen wäre, doch an der einen oder anderen Stelle Kompromisse bzgl. Vollständigkeit und Orthogonalität des Formalismus einzugehen. Allerdings lehrt die Erfahrung doch immer wieder, daß sich derartige Kompromisse eines Tages rächen, wenn man doch einmal mit einem Problem konfrontiert ist, bei dem man eine Kombination von Operatoren benötigt (oder zumindest gewinnbringend einsetzen könnte), die man bis dahin für vollkommen irrelevant gehalten hat.

Aus diesem Grund kann die konsequente Entwicklung eines *vollständigen* und *orthogonalen* Formalismus − einschließlich einer praxistauglichen Implementierung − als ein wesentlicher Beitrag dieser Arbeit betrachtet werden.

6.3 Prozeßalgebren

6.3.1 Einleitung

Neben den im vorangegangenen Abschnitt diskutierten Ausdrucksformalismen, bei denen die Ähnlichkeit zu Interaktionsausdrücken offensichtlich ist, gibt es eine weitere Klasse von Formalismen, meist als *Prozeßalgebren* bezeichnet, bei denen die Verwandtschaft weniger stark ausgeprägt ist. Neben reinen *Beschreibungsmitteln* zur Spezifikation sequentieller und paralleler (typischerweise auch kommunizierender) Prozesse, bieten diese stark mathematisch ausgerichteten Formalismen auch einen Regelapparat zur *Verifikation* von Prozessen bzw. Prozeßsystemen. Implementierungsaspekte spielen

jedoch zumeist nur eine untergeordnete Rolle, obwohl es zum Teil Werkzeuge zum experimentellen Umgang mit einem Formalismus gibt [Cleaveland90, Moller99] (vgl. auch § 6.3.2.3).

Im folgenden soll exemplarisch einer der bekanntesten und einflußreichsten Vertreter von Prozeßalgebren, CSP von Hoare [Hoare85], vorgestellt und mit Interaktionsausdrücken verglichen werden. Andere Formalismen, wie z. B. LOTOS (vgl. § 6.2.4.3), CCS (Calculus of Communicating Systems) [Milner80, Milner89], ATP (Algebraic Theory of Processes) [Hennessy88] oder ACP (Algebra of Communicating Processes) [Baeten90, Bergstra90], sollen hier lediglich erwähnt werden, da ihre Unterschiede zu CSP in Details der Syntax, Semantik, Ausdrucksmächtigkeit etc. für einen Vergleich mit Interaktionsausdrücken nicht wesentlich sind.

6.3.2 CSP (Communicating Sequential Processes)

Analog zur Gliederung der Teilabschnitte von § 6.2, werden im folgenden zunächst die von CSP angebotenen Operatoren vorgestellt (§ 6.3.2.1) sowie kurz auf die Definition der Semantik (§ 6.3.2.2) und die Implementierung des Formalismus (§ 6.3.2.3) eingegangen. Anschließend (§ 6.3.2.4) wird CSP mit Interaktionsausdrücken verglichen, wobei deutlich wird, daß viele der angebotenen Operatoren in beiden Formalismen sehr ähnlich sind, bei genauerer Betrachtung aber doch einige nicht zu vernachlässigende Unterschiede hervortreten.

6.3.2.1 Angebotene Operatoren

Sequentielle Prozesse

CSP besitzt zwei grundlegende Konstrukte zur Beschreibung einfacher sequentieller[8] Prozesse:

- Die *Präfix-Konstruktion* $(x \rightarrow P)$ beschreibt einen Prozeß, der im ersten Schritt die Aktion (oder das Ereignis) x akzeptiert und sich anschließend wie Prozeß P verhält.
- Die *binäre Auswahl* $(x \rightarrow P \mid y \rightarrow Q)$ beschreibt einen Prozeß, der im ersten Schritt sowohl die Aktion x als auch die Aktion y akzeptiert. Abhängig davon, welche dieser Aktionen durch einen externen Benutzer ausgeführt wird, verhält sich der Prozeß anschließend entweder wie Prozeß P oder wie Prozeß Q. Da x und y *verschieden* sein müssen, ist das Verhalten des Prozesses in jedem Fall eindeutig festgelegt.

Sowohl die Präfix-Konstruktion als auch die binäre Auswahl können als Spezialfälle der *allgemeinen Auswahl* $(z\colon B \rightarrow R(z))$ mit einer beliebigen Menge B von Aktionen und einer Funktion $R(z)$, die jedem Wert $z \in B$ einen bestimmten Prozeß zuordnet, aufgefaßt werden:

- Für eine *einelementige* Menge $B = \{ x \}$ und $R(x) = P$ erhält man die Präfix-Konstruktion $(x \rightarrow P)$.
- Für eine *zweielementige* Menge $B = \{ x, y \}$ und $R(x) = P$, $R(y) = Q$ erhält man die binäre Auswahl $(x \rightarrow P \mid y \rightarrow Q)$.
- Für eine *leere* Menge $B = \emptyset$ schließlich erhält man den speziellen Prozeß STOP, der zwar nichts tut, aber formal als „Verankerung“ gebraucht wird, da auf der rechten Seite einer Präfix-Konstruktion $(x \rightarrow P)$ immer ein *Prozeß* (und keine Aktion) stehen muß.

Iterative Prozesse können mit Hilfe von *Rekursionsgleichungen* definiert werden: Wenn $F(X)$ ein Ausdruck der Gestalt $(z\colon B \rightarrow R(z, X))$ ist (d. h. mit einem Präfix beginnt), der die Prozeßvariable X enthält, dann bezeichnet $\mu X.\, F(X)$ die (in diesem Fall eindeutige) Lösung der Rekursionsgleichung $X = F(X)$. Beispielsweise läßt sich ein Prozeß, dessen Verhalten in etwa dem Interaktionsausdruck $\ominus\,(a - b)$ entspricht, als Lösung der Rekursionsgleichung $X = (a \rightarrow (b \rightarrow X))$ definieren.

[8] Die Bezeichnung „sequentiell“ wird hier einfach im Sinne von „nicht parallel“ gebraucht. Im Gegensatz hierzu verbindet Hoare mit dem Begriff *sequential process* bestimmte zusätzliche Eigenschaften, die weiter unten beschrieben werden.

Diese Vorgehensweise läßt sich auf ein *System* von mehreren Rekursionsgleichungen erweitern, und durch die Verwendung von Indizes lassen sich sogar Systeme mit *unendlich vielen* Gleichungen formulieren, wie z. B.:

$P_0 = (\text{produce} \to P_1)$,

$P_n = (\text{produce} \to P_{n+1} \mid \text{consume} \to P_{n-1})$ für $n \geq 1$.

Durch dieses System von Gleichungen wird ein Prozeß $P = P_0$ definiert, der die Aktionen produce und consume in beliebiger Reihenfolge akzeptiert, sofern die Anzahl der consumes die Anzahl der produces nicht übersteigt. Dasselbe Verhalten kann daher auch durch den Interaktionsausdruck ◎ (produce – consume) beschrieben werden (vgl. § 2.3.1.2).

Nebenläufigkeit

Zwei beliebige Prozesse P und Q lassen sich mittels des *Interaction-* oder *Concurrency-Operators* $\|$ zu einem neuen Prozeß $P \| Q$ kombinieren. Die Semantik entspricht hierbei genau dem *IAA-Synchronisationsoperator*: Sofern die Alphabete von P und Q gleich sind, müssen alle Aktionsausführungen von P und Q synchron erfolgen, bei teilweise überlappenden Alphabeten müssen lediglich die gemeinsamen Aktionen synchron ausgeführt werden, und bei disjunkten Alphabeten können P und Q vollkommen unabhängig voneinander ausgeführt werden.

Im Gegensatz zu Interaktionsausdrücken, bei denen das Alphabet $\alpha(x)$ eines Ausdrucks x implizit durch den Ausdruck selbst gegeben ist (vgl. § 3.3.1.9), kann es bei CSP nach Belieben um zusätzliche Aktionen erweitert werden.

Nichtdeterminismus

Im Gegensatz zu Interaktionsausdrücken erlauben die bisher beschriebenen Konstrukte keinerlei „Mehrdeutigkeiten“: Bei Ausführung einer Aktion a ist stets eindeutig bestimmt, um welches Vorkommen von a in der Prozeßbeschreibung es sich handelt. Insbesondere müssen die Aktionen x und y bei einer Auswahl $(x \to P \mid y \to Q)$ *verschieden* sein, und auch der Concurrency-Operator $\|$ ist − im Gegensatz zur parallelen Komposition bei Interaktionsausdrücken − so definiert, daß keine Mehrdeutigkeiten auftreten können: Wenn P und Q gemeinsame Aktionen enthalten, müssen sie *synchron* ausgeführt werden.

Zur Beschreibung „mehrdeutiger“ oder *nichtdeterministischer* Prozesse bietet CSP die beiden Operatoren $\sqcap$ und $[]$ an.

Für zwei beliebige Prozesse P und Q beschreibt $P \sqcap Q$ einen Prozeß, der sich *entweder* wie P *oder* wie Q verhalten kann. Weder der Zeitpunkt der Entscheidung für die eine oder die andere Alternative noch die Faktoren, die diese Entscheidung möglicherweise beeinflussen, sind nach außen hin bekannt. Beispielsweise könnte sich ein Implementierer des Prozesses $P \sqcap Q$ bereits bei der Implementierung für eine der beiden Alternativen entscheiden, d. h. entweder nur Prozeß P oder nur Prozeß Q implementieren. Er könnte aber auch beide Alternativen bereitstellen und eine zufällige oder von bestimmten Kriterien abhängige Auswahl zur Laufzeit vorsehen usw. Insbesondere *könnte* sich der Prozeß $P \sqcap Q$ auch so verhalten wie der Interaktionsausdruck $P \bigcirc Q$, d. h. er könnte die Entscheidung für P oder Q solange „aufschieben“, bis eine Aktion ausgeführt wird, die eindeutig einer der beiden Alternativen zugeordnet werden kann. Das *exakte* Verhalten des Prozesses $P \sqcap Q$ ist also *nicht vorhersagbar*.

Der Operator $[]$ stellt eine „partiell nichtdeterministische“ Variante des Operators $\sqcap$ dar: Ein Prozeß $P \, [] \, Q$ verhält sich wie Prozeß P (bzw. Q), wenn die erste ausgeführte Aktion *eindeutig* diesem Prozeß zugeordnet werden kann; andernfalls, d. h. wenn die erste Aktion sowohl von P als auch von Q akzeptiert wird, kann die Auswahl zwischen P und Q wiederum *willkürlich* getroffen werden.

Beispiel

Das folgende Beispiel verdeutlicht sowohl den feinen Unterschied zwischen den CSP-Operatoren $\sqcap$ und $[]$ als auch ihren Unterschied zum IAA-Disjunktionsoperator $\bigcirc$.

Die Prozesse

$$P \equiv (a \rightarrow a \rightarrow \mathsf{STOP} \mid b \rightarrow a \rightarrow \mathsf{STOP}) \quad \text{und} \quad Q \equiv (b \rightarrow c \rightarrow \mathsf{STOP} \mid c \rightarrow c \rightarrow \mathsf{STOP})$$

entsprechen direkt den Interaktionsausdrücken

$$X \equiv (a - a) \bigcirc (b - a) \quad \text{bzw.} \quad Y \equiv (b - c) \bigcirc (c - c).$$

Der Prozeß $P \mathbin{[]} Q$ akzeptiert im ersten Schritt (ebenso wie der Interaktionsausdruck $X \bigcirc Y$) mit Sicherheit die Aktionen a, b und c. Bei Ausführung von a (bzw. c) verhält er sich anschließend garantiert wie Prozeß P (bzw. Q), d. h. er akzeptiert anschließend (ebenso wie der Ausdruck $X \bigcirc Y$) ein weiteres a (bzw. c). Bei Ausführung von b jedoch, das sowohl von Prozeß P als auch von Prozeß Q „konsumiert“ werden kann, wird *unter Umständen* eine willkürliche Auswahl zwischen P und Q getroffen, d. h. anschließend akzeptiert der Prozeß $P \mathbin{[]} Q$ *möglicherweise* nur noch *a oder c*. (Je nach Implementierung könnte er aber auch beides akzeptieren.) Im Gegensatz hierzu schiebt der Interaktionsausdruck $X \bigcirc Y$ nach Ausführung von b die Entscheidung zwischen X und Y *auf jeden Fall* auf, so daß er anschließend mit Sicherheit *a und c* akzeptiert.

Beim Prozeß $P \sqcap Q$ schließlich kann bereits *am Anfang* eine willkürliche Auswahl zwischen P und Q getroffen werden, so daß er als erste Aktion *möglicherweise* nur a und b *oder* nur b und c akzeptiert. (Je nach Implementierung könnte er aber auch alle drei akzeptieren.)

Interleaving

Auf der Basis des Operators $[]$ läßt sich eine als *Interleaving* bezeichnete Variante $|||$ des Concurrency-Operators $||$ definieren, die im Prinzip einer „partiell nichtdeterministischen“ *parallelen Komposition* $\circledcirc$ entspricht: Solange eine Aktion *eindeutig* einem der beiden Prozesse P oder Q zugeordnet werden kann, ist das Verhalten von $P \mathbin{|||} Q$ deterministisch. Andernfalls kann willkürlich entschieden werden, welcher Prozeß die Aktion „konsumiert“, was natürlich wesentlichen Einfluß auf das weitere Verhalten von $P \mathbin{|||} Q$ haben kann. Der Interaktionsausdruck $P \circledcirc Q$ hingegen schiebt in diesem Fall die Entscheidung auf, ob die Aktion von P oder von Q konsumiert wird, d. h. er verfolgt anschließend zwei entsprechende Alternativen.

Sequentielle Komposition

Überraschenderweise gibt es unter den bisher eingeführten Operatoren noch keinen, der die *sequentielle Komposition* zweier beliebiger *Prozesse* erlaubt: Die Präfix-Konstruktion $(x \rightarrow P)$ erlaubt nur auf der rechten Seite einen Prozeß, während auf der linken Seite eine einzelne Aktion stehen muß!

Die allgemeine sequentielle Komposition $P;Q$ wird erst eingeführt, nachdem unter Zuhilfenahme eines speziellen Aktionssymbols ✔ der Begriff eines *erfolgreich terminierenden Prozesses* definiert ist.[9] Außerdem werden noch verschiedene Arten von *Interrupts* definiert, die alle auf der Grundform $P \wedge Q$ basieren, die besagt, daß die Ausführung des Prozesses P jederzeit durch die Ausführung des Prozesses Q unterbrochen (im Sinne von *abgebrochen*) werden kann.

6.3.2.2 Definition der Semantik

Sowohl für deterministische als auch für nichtdeterministische Prozesse werden präzise, mathematisch fundierte Definitionen gegeben. Ein *deterministischer* Prozeß wird beispielsweise eindeutig durch sein *Alphabet* und die Menge seiner *Spuren* (engl. traces) charakterisiert, wobei eine Spur

[9] Derartige Prozesse werden von Hoare als *sequential processes* bezeichnet.

nichts anderes als ein *partielles Wort* im Sinne einer sprachtheoretischen Semantik darstellt. Zur Definition von Prozessen mittels Rekursionsgleichungen wird eine *Fixpunkttheorie* verwendet, mit deren Hilfe sowohl die *Existenz* als auch die *Eindeutigkeit* von Lösungen gezeigt werden kann, sofern die Gleichungen gewissen, syntaktisch nachprüfbaren Bedingungen genügen.

Im Fall *nichtdeterministischer* Prozesse werden zusätzliche Charakteristika von Prozessen, wie z. B. *refusals*, *failures* und *divergences* betrachtet, deren Erläuterung hier zu weit führen würde. Interessant ist jedoch, daß im Kontext nichtdeterministischer Prozesse die oben erwähnten Bedingungen für Rekursionsgleichungen aufgehoben werden können: Sofern eine gegebene Gleichung von mehr als einem Prozeß erfüllt wird, wird derjenige Prozeß als „eindeutige" Lösung der Gleichung definiert, der *am wenigsten deterministisch* ist und damit im Prinzip eine Art „Obermenge" aller übrigen Lösungen darstellt. Beispielsweise erhält man als „eindeutige Lösung" der trivialen Gleichung $X = X$ den Prozeß CHAOS, dessen Verhalten − wie der Name schon andeutet − absolut willkürlich sein kann.

6.3.2.3 Implementierung des Formalismus

Hoare skizziert für sämtliche CSP-Operatoren eine überraschend einfache und elegante Implementierung in einer *funktionalen Programmiersprache* wie z. B. LISP. Selbst (Systeme von unendlich vielen!) Rekursionsgleichungen lassen sich relativ direkt in entsprechende Funktionsdefinitionen umsetzen. Auf diese Weise erhält man unmittelbar eine einfache *Testumgebung*, in der man das dynamische Verhalten von Prozessen experimentell untersuchen kann. Da bei dieser Implementierung jedoch keinerlei Wert auf Laufzeit-Effizienz gelegt wird, ist sie für einen realen Einsatz sicherlich nicht geeignet.

Anders verhält es sich mit Abbildungen von CSP auf bestimmte *parallele Programmiersprachen*, wie z. B. Ada oder Occam, die geeignete Konstrukte zur Formulierung von *Nebenläufigkeit*, *Nichtdeterminismus* und *Kommunikation* bereitstellen. Insbesondere *Occam* [May83, Schütte88] lehnt sich auch syntaktisch sehr stark an CSP an, so daß man es durchaus als Implementierung von CSP bezeichnen könnte. Im Vergleich zu modernen strukturierten Programmiersprachen ist Occam jedoch eher als „Assembler der parallelen Programmierung" zu bezeichnen [Bal89].

6.3.2.4 Vergleich mit Interaktionsausdrücken

CSP ist zweifellos ein sehr mächtiger formaler Apparat, dessen Möglichkeiten z. T. weit über die von Interaktionsausdrücken hinausgehen. Neben den hier skizzierten Konzepten hat man im Prinzip eine vollständige parallele Programmiersprache mit einer präzisen formalen Semantik sowie zugehörigen Verifikationsmöglichkeiten zur Verfügung.

Auf den ersten Blick könnte daher der Eindruck entstehen, CSP sei eine echte *Obermenge* von Interaktionsausdrücken: Es gibt Operatoren für sequentielle und parallele *Komposition* (Semikolon oder $\rightarrow$ und |||) *Disjunktion* (|, $\sqcap$ oder []) und *Konjunktion* (||, sofern die Alphabete der beiden Operanden gleich sind, was man prinzipiell immer gewährleisten kann), und Rekursionsgleichungen können u. a. zur Definition von *Iterationen* verwendet werden. Bei genauerer Betrachtung ergeben sich jedoch mindestens zwei wesentliche Unterschiede in der Grundkonzeption der beiden Formalismen, die dazu führen, daß CSP für den Kontext dieser Arbeit *nicht* anwendbar ist.

Mehrdeutigkeit gegenüber Nichtdeterminismus

Der erste wesentliche Unterschied besteht darin, daß Interaktionsausdrücke zwar *mehrdeutig* sein können (in dem Sinn, daß im Moment einer Aktionsausführung u. U. noch nicht entschieden werden kann, um welches Vorkommen der Aktion im Ausdruck es sich handelt), sich aber nichtsdestotrotz immer *deterministisch* (im Sinne von *vorhersagbar*) verhalten (vgl. § 2.4.3). Man beachte, daß sich in diesem Sinne auch ein nichtdeterministischer endlicher Automat deterministisch (d. h. vorhersagbar) verhält: Eine Eingabe wird genau dann akzeptiert, wenn es mindestens einen mit dieser Eingabe markierten Pfad vom Anfangs- zu einem Endzustand des Automaten gibt [Hopcroft90, Schöning95].

Im Gegensatz hierzu kann sich ein CSP-Prozeß wirklich *nichtdeterministisch* (im Sinne von *nicht exakt vorhersagbar*) verhalten, was für Anwendungsbereiche wie Workflow-Management nicht akzeptabel wäre. Auf der anderen Seite läßt sich der IAA-Disjunktionsoperator $\bigcirc$ nicht exakt auf CSP abbilden: Bei der Abbildung auf das Auswahl-Konstrukt $(x \rightarrow P \mid y \rightarrow Q)$ verliert man die Möglichkeit der „kontrollierten Mehrdeutigkeit" (x und y müssen verschieden sein), während man bei der Abbildung auf einen der Operatoren $\sqcap$ oder $[]$ die „Vorhersagbarkeit" verliert. Das für Interaktionsausdrücke wesentliche Prinzip der *Souveränität des Benutzers* (vgl. § 2.4.3.5), das von Hoare als *angelic nondeterminism* bezeichnet wird, wird in CSP so nicht unterstützt. Da Mehrdeutigkeiten nicht nur durch Disjunktionen hervorgerufen werden können, sondern z. B. auch durch Iterationen oder parallele Kompositionen, hat dieser Unterschied weitreichende Konsequenzen, insbesondere auch für die operationale Semantik und die Implementierung des Formalismus.

Anmerkung: Würde man von einem Prozeß $P \sqcap Q$ oder $P \; [] \; Q$ nur die Menge seiner Spuren (d. h. seine Sprache) betrachten, so wäre diese für beide Prozesse tatsächlich identisch zur Menge der partiellen Worte des Interaktionsausdrucks $P \bigcirc Q$. Allerdings würde man bei dieser Sichtweise einen großen Teil der Theorie von CSP, nämlich gerade die umfassende Behandlung nichtdeterministischer Prozesse, über Bord werfen. Der verbleibende Rest wäre, was die formale Semantik betrifft, zweifellos sehr ähnlich zu Interaktionsausdrücken. Wie bereits mehrfach erwähnt, besteht ein wesentlicher Beitrag dieser Arbeit jedoch darin, diese formale Semantik geeignet „mit Leben zu füllen", d. h. eine effiziente Implementierung des Formalismus bereitzustellen, die die Souveränität des Benutzers respektiert, d. h. an keiner Stelle nichtdeterministische Entscheidungen trifft. Für diese Aufgabe bieten Theorien wie CSP jedoch keinerlei Unterstützung, weil sie Prozesse wie $P \sqcap Q$ oder $P \; [] \; Q$ einfach nichtdeterministisch implementieren, was vergleichsweise trivial ist.

Terminationsverhalten

Ein weiterer wesentlicher Unterschied zwischen CSP und Interaktionsausdrücken besteht im *Terminationsverhalten* von Prozessen bzw. Ausdrücken. Während ein Prozeß in CSP vollkommen autonom entscheiden muß, wann er terminiert, muß ein Interaktionsausdruck lediglich signalisieren, daß er *terminationsbereit* ist (was genau dann der Fall ist, wenn er ein vollständiges Wort verarbeitet hat, d. h. wenn sein aktueller Zustand ein *Endzustand* ist), und kann die endgültige Terminationsentscheidung dem syntaktisch übergeordneten Ausdruck überlassen.

Dieser Unterschied ist vor allem im Kontext von Wiederholungen von Bedeutung. Prinzipiell läßt sich eine sequentielle Iteration $*P$ (mit einem beliebigen Prozeß P) in CSP als Lösung der Rekursionsgleichung $X = P; X$ definieren. Allerdings ist dieser Prozeß $*P$ dann mit der schwierigen Frage konfrontiert, wann er terminieren soll, und die einzig konsequente und sinnvolle Antwort darauf lautet, daß er nie terminiert. Diese Antwort ist allerdings unbefriedigend, wenn $*P$ in einen übergeordneten Prozeß wie z. B. $*P; Q$ eingebettet ist, weil dieser dann faktisch äquivalent zu $*P$ ist: Weil $*P$ nie terminiert, kommt Q nie zur Ausführung.

Der entsprechende Interaktionsausdruck $\ominus P$ hingegen löst dieses Problem elegant durch „Delegation" und „Teamwork": Er muß lediglich nach jedem Iterationsschritt seine Terminationsbereitschaft signalisieren, was einen übergeordneten Ausdruck wie z. B. $\ominus P - Q$ dazu veranlaßt, im nächsten Schritt sowohl die initialen Aktionen von P als auch die von Q zu akzeptieren und damit dem *Benutzer* die Entscheidung über die Termination oder Fortsetzung der Iteration $\ominus P$ zu überlassen.

Sofern die Mengen der initialen Aktion von P und Q disjunkt sind, kann man einen Prozeß mit demselben Verhalten prinzipiell auch in CSP definieren, nämlich als Lösung der Rekursionsgleichung $X = Q \; [] \; (P; X)$ [Brookes84]. Diese Lösung ist jedoch in zweierlei Hinsicht unbefriedigend: Zum einen ist das Verhalten des Prozesses nichtdeterministisch, sobald P und Q gemeinsame initiale Aktionen besitzen, zum anderen ist die Definition der Iteration mit dem konkreten Nachfolgerprozeß Q verwoben, d. h. eine *unabhängige* Definition der Iteration ist nicht möglich.

Ein letzter Versuch definiert $*P$ als Lösung der Rekursionsgleichung $X = P; (X \sqcap \mathsf{SKIP})$, die besagt, daß nach Ausführung von P entweder eine neue Iteration X durchlaufen wird oder der Prozeß

erfolgreich terminiert.[10] Allerdings besitzt auch diese Lösung den Nachteil, daß der Prozeß X nichtdeterministisch ist: Im Gegensatz zum Interaktionsausdruck ⊖ P kann er eigenmächtig und willkürlich nach einer beliebigen Anzahl von Iterationen terminieren, ohne hierbei die Wünsche des Benutzers zu respektieren.

Ähnliche Schwierigkeiten ergeben sich auch bei der Definition von Prozessen, die den Interaktionsausdrücken ⌣ P und ◎ P entsprechen. (Bei letzterem kommt auch noch das oben diskutierte Problem von Mehrdeutigkeit gegenüber Nichtdeterminismus hinzu.) Dies bedeutet, daß auch ein zweites wesentliches Prinzip von Interaktionsausdrücken, die *modulare Kombination* von Ausdrücken (vgl. § 2.8.2.4), von CSP nicht durchgängig unterstützt wird.

Sonstiges

Abgesehen von diesen beiden wesentlichen Unterschieden zwischen CSP und Interaktionsausdrücken, sollte erwähnt werden, daß auch CSP *nicht vollständig orthogonal* ist, da auf der linken Seite des Präfixoperators $\rightarrow$ kein Prozeß, sondern nur eine Aktion stehen darf. Außerdem kann der Auswahloperator | nur in Kombination mit dem Präfixoperator $\rightarrow$ verwendet werden.

Desweiteren wurde in § 2.8.2.5 erläutert, daß *explizite Iterationsoperatoren*, wie sie von Interaktionsausdrücken angeboten werden, gegenüber implizit durch Rekursionsgleichungen formulierten Iterationen den Vorteil besitzen, daß sie kompakter und direkter die intendierte Semantik zum Ausdruck bringen und daher − insbesondere für mathematisch ungeübte Anwender − besser verständlich sind.

6.4 Erweiterte Transaktionsmodelle

Sobald man klassische ACID-Transaktionen [Gray81, Agrawal92] dahingehend erweitert, daß ein Transaktionsmanager nicht nur einfache Lese- und Schreiboperationen auf Datenobjekten kennt, deren Synchronisationsbedingungen a priori festgelegt sind, so ist man mit dem Problem konfrontiert, die zulässigen Ausführungsreihenfolgen semantisch höherer Operationen − wie z. B. Belastung oder Gutschrift eines Geldbetrags auf einem Konto − geeignet zu spezifizieren. Ähnliche Probleme ergeben sich, wenn man komplexere Transaktionsstrukturen − wie z. B. offen oder geschlossen geschachtelte Transaktionen [Moss81, Weikum92] − flexibel modellieren will, da hierfür spezifiziert werden muß, in welcher Reihenfolge Aktionen wie das Starten, Beenden oder Abbrechen von Teiltransaktionen ausgeführt werden dürfen.

Daher wurden auch im Kontext erweiterter Transaktionsmodelle [Elmagarmid92, DEB93] immer wieder Formalismen vorgeschlagen, mit denen − ebenso wie mit Interaktionsausdrücken − zulässige Ausführungsreihenfolgen von Aktionen beschrieben werden können. Aufgrund seines relativ hohen Bekanntheitsgrades und seiner „terminologischen Verwandtschaft“ zu Inter-Workflow-Abhängigkeiten, wird im folgenden zunächst der Ansatz der *inter-task dependencies* vorgestellt und mit Interaktionsausdrücken verglichen (§ 6.4.1). Anschließend werden exemplarisch zwei weitere Ansätze diskutiert, bei denen grundsätzlich ähnliche Ideen wie bei der Entwicklung von Interaktionsausdrücken verfolgt wurden (§ 6.4.2).

6.4.1 Inter-task dependencies

6.4.1.1 Angebotene Operatoren

In [Klein91] wurden erstmals zwei grundlegende Operatoren zur Beschreibung von Abhängigkeiten in verteilten Systemen vorgeschlagen, die in verschiedenen Folgearbeiten [Attie93, Tang95] zur Spezifikation von *inter-task dependencies* verwendet werden:

[10] SKIP ist (ähnlich wie STOP) ein Prozeß, der nichts tut, aber (anders als STOP) erfolgreich terminiert.

- Die *Reihenfolgebeziehung* $e_1 < e_2$ legt fest, daß das Ereignis e_1 *vor* dem Ereignis e_2 eintreten muß, sofern *beide* Ereignisse tatsächlich eintreten.
- Die *Existenzabhängigkeit* $e_1 \rightarrow e_2$ entspricht einer logischen Implikation: *Wenn* das Ereignis e_1 eintritt, dann *muß* auch das Ereignis e_2 eintreten. Über die Reihenfolge von e_1 und e_2 wird jedoch keine Aussage gemacht.

Zur Beschreibung komplexerer Abhängigkeiten können diese Operatoren nach Belieben kombiniert und durch logische Operatoren verknüpft werden. Ein komplexer Ausdruck E wird dann ebenfalls als Ereignis interpretiert, das in dem Moment eintritt, in dem der Ausdruck E (als logisches Prädikat) wahr wird. Beispielsweise beschreibt der Ausdruck $e_1 \rightarrow (e_2 < e_3)$, daß die Reihenfolgebeziehung $e_2 < e_3$ eingehalten werden muß, wenn das Ereignis e_1 auftritt. Durch die Verwendung von Abstraktionsmechanismen (Makros) können komplexe und schwer verständliche Ausdrücke benutzerfreundlich „verpackt“ werden.

6.4.1.2 Definition der Semantik

Während in der Originalarbeit [Klein91] keine Aussagen zur formalen Semantik der Operatoren gemacht werden, werden sie in [Attie93] auf Formeln der temporalen Aussagenlogik CTL (computation tree logic) [Emerson90] abgebildet, auf die an dieser Stelle nicht näher eingegangen werden soll.

Unabhängig davon werden in [Tang95] verschiedene Klassen von Ereignisfolgen definiert, u. a. Ereignisfolgen, die bezüglich eines gegebenen Ausdrucks *durchführbar* (engl. feasible) sind. Diese entsprechen in gewisser Weise den vollständigen Worten eines Interaktionsausdrucks.

6.4.1.3 Implementierung des Formalismus

Zur Einhaltung der spezifizierten Bedingungen können verschiedene Strategien verfolgt werden, je nachdem welche *Eigenschaften* (oder „Attribute“) die einzelnen Ereignisse besitzen.[11]

Beispielsweise kann die Einhaltung der Reihenfolgebeziehung $e_1 < e_2$ erzwungen werden, indem man das Auftreten von e_2 solange *verzögert*, bis das Ereignis e_1 entweder eingetreten ist oder mit Sicherheit nicht mehr eintreten wird. Dies setzt jedoch voraus, daß das Ereignis e_2 *verzögerbar* (engl. delayable) ist und daß man ggf. zuverlässig weiß, daß e_1 nicht mehr eintreten kann (z. B. weil der *Agent*, der e_1 ausführen kann, bereits terminiert ist).

Die Existenzabhängigkeit $e_1 \rightarrow e_2$ kann eingehalten werden, indem man e_1 solange verzögert, bis e_2 eingetreten ist (ggf. muß man e_1 *zurückweisen*, wenn e_2 definitiv nicht mehr eintreten wird), oder indem man e_2 *erzwingt*, nachdem e_1 aufgetreten ist. Letztere Strategie setzt jedoch voraus, daß das Ereignis e_2 *erzwingbar* (engl. enforceable) ist. Falls weder e_1 verzögerbar noch e_2 erzwingbar ist, kann die Einhaltung der Bedingung $e_1 \rightarrow e_2$ nicht gewährleistet werden.

In [Attie93] und [Tang95] werden verschiedene Möglichkeiten zur automatischen Generierung von Zustandsautomaten beschrieben, die jeweils einen gegebenen Ausdruck implementieren und mit Hilfe eines *enforcement protocols* dafür sorgen, daß die durch ihn spezifizierte Bedingung bei der Ausführung von Aktionen eingehalten wird (sofern dies prinzipiell möglich ist; siehe oben).

6.4.1.4 Vergleich mit Interaktionsausdrücken

Obwohl gelegentlich behauptet (oder zumindest „suggeriert“) wird, daß die beiden Basisoperatoren $<$ und $\rightarrow$ zur Spezifikation *beliebiger* Abhängigkeitsbeziehungen ausreichend sind [Tang95, Jablonski95a], lassen sich mit ihnen offensichtlich keine Iterationen beschreiben: Jedes Ereignis e eines Ausdrucks E kann in einer zulässigen Ereignisfolge *höchstens einmal* auftreten [Attie93, Faase96]. Im Anwendungskontext erweiterter Transaktionsmodelle, wo typische Ereignisse im Starten, Beenden und Abbrechen von Teiltransaktionen bestehen (die im Rahmen einer Gesamttransaktion nur einmal ausgeführt werden), mag dies ausreichend sein. Zur Synchronisation von Workflows, bei denen so-

[11] Wie in § 2.6.3 erläutert, entspricht die Bezeichnung *Ereignis* (engl. event) hier nicht unbedingt der natürlichen Bedeutung dieses Begriffs.

wohl einzelne Aktivitäten als auch ganze Arbeitsabläufe wiederholt ausgeführt werden können, sind Operatoren zur Beschreibung von Iterationen jedoch essentiell (vgl. die Beispiele in § 2.7 bzw. § 5.4).

Ein weiterer wesentlicher Unterschied besteht darin, daß die Ausführung einer Aktion bzw. Aktivität eines Workflows normalerweise nicht *erzwungen* werden kann, was die Anzahl der Strategien zur Einhaltung der Bedingungen u. U. erheblich einschränkt. Insbesondere kann eine Existenzabhängigkeit $e_1 \rightarrow e_2$ nur dadurch eingehalten werden, daß das Eintreten von e_1 solange verzögert (d. h. verboten) wird, bis e_2 tatsächlich eingetreten ist, d. h. $e_1 \rightarrow e_2$ stellt dann faktisch eine Reihenfolgebeziehung $e_1 > e_2$ dar, die sich nur geringfügig von der Bedingung $e_2 < e_1$ unterscheidet. (Letztere erlaubt das Eintreten von e_1 auch dann, wenn e_2 nicht mehr eintreten kann.)

Schließlich sollte zur Vermeidung von „terminologischen Mißverständnissen“ erwähnt werden, daß eine *task* im Sinne von [Attie93, Tang95] nicht etwa einem Workflow, sondern vielmehr einer einzelnen *Aktivität* bzw. einem Workflow-*Schritt* entspricht. Demzufolge sind mit *inter-task dependencies* Abhängigkeiten zwischen Workflowschritten, also primär *Intra-* statt Inter-Workflow-Abhängigkeiten gemeint.

6.4.2 Weitere Ansätze

Aus der großen Fülle anderer erweiterter Transaktionsmodelle sollen im folgenden lediglich zwei Ansätze herausgegriffen werden, bei denen grundsätzlich ähnliche Ideen zur Beschreibung von Reihenfolgebeziehungen wie bei der Entwicklung von Interaktionsausdrücken verfolgt wurden.

In [Rastogi93] werden *reguläre Ausdrücke* zur Spezifikation *unzulässiger* Ausführungsreihenfolgen von Teiltransaktionen verwendet, während in [Nodine92] *LR(0)-Grammatiken* zur Beschreibung zulässiger *und* unzulässiger Reihenfolgen eingesetzt werden. Obwohl die Formulierung *zulässiger* Ausführungsreihenfolgen aus Anwendersicht grundsätzlich einfacher und „natürlicher“ zu sein scheint, mag die Entscheidung, im ersten Ansatz stattdessen *unzulässige* Reihenfolgen zu spezifizieren, durch spezielle anwendungsorientierte Rahmenbedingungen begründet sein. Die Spezifikation *beider* Arten von Reihenfolgen im zweiten Ansatz erscheint jedoch redundant, insbesondere da sich die exemplarisch vorgestellten „negativen“ Grammatiken (zur Spezifikation unzulässiger Reihenfolgen) jeweils unmittelbar aus den zugehörigen „positiven“ (zur Spezifikation zulässiger Reihenfolgen) ableiten lassen.

Bezüglich Ausdrucksmächtigkeit stößt man mit regulären Ausdrücken zweifellos sehr schnell an Grenzen, während man mit LR(0)-Grammatiken prinzipiell sämtliche *deterministisch kontextfreien Sprachen* beschreiben kann [Nodine92, Hopcroft90]. Sie stellen daher einen guten Kompromiß zwischen möglichst hoher Ausdrucksmächtigkeit auf der einen Seite und effizienter Implementierbarkeit auf der anderen Seite dar. Außerdem besitzen sie, im Gegensatz zu allgemeineren LR(k)-Grammatiken die angenehme Eigenschaft, daß ein Wort zeichenweise, ohne Verwendung von *Look-ahead-Symbolen*, analysiert werden kann. Allerdings fehlt ihnen das im Kontext dieser Arbeit wesentliche Konzept der *Parallelität*, also insbesondere die häufig benötigte parallele Iteration sowie parallele Quantoren (vgl. z. B. § 2.7.5).

6.5 Petrinetze

Petrinetze [Peterson77, Reisig86, Baumgarten96] sind ein weit verbreiteter und vielseitig einsetzbarer Formalismus zur Spezifikation nebenläufiger Systeme. Obwohl sie im Gegensatz zu Interaktionsausdrücken keine spezifischen *Operatoren* (wie z. B. – oder ◎ für sequentielle oder parallele Komposition) anbieten, können die mit diesen Operatoren assoziierten *Konzepte* (z. B. sequentielle oder parallele Ausführung von Aktionen) dennoch modelliert werden. Außerdem besitzen sie rein äußerlich gewisse Ähnlichkeiten mit Interaktions*graphen*.

Aufgrund dieser konzeptuellen und optischen Verwandtschaft wird im folgenden (§ 6.5.1) zunächst untersucht, inwieweit sich Interaktionsgraphen auf einfache *Stellen-Transitionen-Netze* (§ 6.5.1.1 bis § 6.5.1.5) oder auf *Netze mit individuellen Marken* (§ 6.5.1.6 und § 6.5.1.7) abbilden lassen. Anschließend (§ 6.5.2) wird diese Transformation, die teilweise zwar schwierig, prinzipiell aber durchführbar

erscheint, kritisch hinterfragt. Ähnlich wie bei Prozeßalgebren ergibt sich hierbei, daß Petrinetze einige für die Entwicklung von Interaktionsgraphen wesentliche Prinzipien nicht zufriedenstellend unterstützen und daher zur Beschreibung von Inter-Workflow-Abhängigkeiten nicht geeignet sind.

6.5.1 Transformation von Interaktionsgraphen in Petrinetze

6.5.1.1 Stellen-Transitionen-Netze

Wie bereits erwähnt, besitzen Interaktionsgraphen − sowohl was ihre äußere Form als auch ihre inhaltliche Zielsetzung betrifft − durchaus Ähnlichkeiten mit Petrinetzen. Betrachtet man z. B. den Graphen in Abb. 6.9, so könnte man ihn einerseits als graphische Repräsentation des Interaktionsausdrucks $a \bigcirc b$ und andererseits als einfaches *Stellen-Transitionen-Netz* auffassen, in dem Stellen wie üblich durch Kreise und Transitionen durch beschriftete Rechtecke dargestellt sind.[12] Außerdem soll angenommen werden, daß Kanten ohne Pfeile implizit von links nach rechts gerichtet sind.

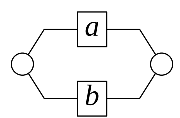


Abbildung 6.9: Entweder-oder-Verzweigung als Stellen-Transitionen-Netz

In diesem einfachen Beispiel besitzt der Graph sogar in beiden Interpretationen dieselbe *Semantik*: Betrachtet man ihn als Interaktionsgraphen, so kann man sich einen Läufer vorstellen, der − ausgehend vom linken ○-Knoten − eine der Aktionen *a* oder *b* durchläuft und sich anschließend am rechten ○-Knoten befindet. Bei einer Interpretation als Petrinetz verwendet man anstelle eines Läufers eine *Marke*, die sich am Anfang auf der linken Stelle befinden möge (vgl. Abb. 6.10, links). In der so definierten Ausgangssituation kann genau eine der beiden Transitionen *a* oder *b* *schalten*, was zur Folge hat, daß die Marke von der linken Stelle abgezogen und auf die rechte Stelle gelegt wird (vgl. Abb. 6.10, rechts). Somit spezifiziert das Petrinetz mit der vereinbarten Anfangsmarkierung dieselbe Bedingung wie der Interaktionsgraph: Es darf genau eine der beiden Aktionen *a* oder *b* ausgeführt werden.

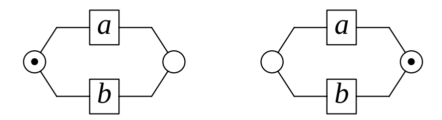


Abbildung 6.10: Schaltverhalten des Stellen-Transitionen-Netzes

6.5.1.2 Hilfsstellen

Auch der Interaktionsgraph in Abb. 6.11 (links) kann ohne große Mühe als Petrinetz interpretiert werden, wenn man zwischen je zwei aufeinanderfolgenden Aktionen bzw. Transitionen eine zusätzliche *Hilfsstelle* einfügt und wie oben eine geeignete Anfangsmarkierung vereinbart (Abb. 6.11, rechts).

[12] Strenggenommen handelt es sich um ein *etikettiertes Petrinetz* [Baumgarten96] (engl. *labeled Petri net* [Peterson77]), das neben Stellen, Transitionen und Kanten eine Abbildung $\lambda: T \to \Sigma$ von Transitionen $t \in T$ auf Aktionen $a \in \Sigma$ besitzt. Jede Transition t wird mit der Aktion $\lambda(t)$ beschriftet.



Abbildung 6.11: Verwendung von Hilfsstellen

Allerdings treten in diesem Beispiel bereits gewisse Probleme bzgl. der exakten Interpretation des Petrinetzes auf: Betrachtet man nur die zugehörige *Petrinetz-Sprache*, d. h. die Menge aller möglichen *Schaltfolgen* des Netzes,[13] so entspricht diese unmittelbar der Menge

$\Psi((a - b) \bigcirc (a - c)) = \{ \langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, c\rangle \}$ bzw.
$\Phi((a - b) \bigcirc (a - c)) = \{ \langle a, b\rangle, \langle a, c\rangle \}$

des korrespondierenden Interaktionsausdrucks $(a - b) \bigcirc (a - c)$, je nachdem ob man beliebige oder nur bestimmte Markierungen des Netzes als zulässige *Endmarkierungen* betrachtet. Versucht man jedoch, das Netz Schritt für Schritt *auszuführen*, indem man jeweils eine aktivierte Transition, die einer in der realen Welt ausgeführten Aktion entspricht, schalten läßt, so ist man unmittelbar mit den in § 2.4.3 diskutierten *Entscheidungsproblemen* konfrontiert. So kann im initialen Zustand von Abb. 6.11 sowohl die obere als auch die untere Transition *a* schalten, und abhängig davon, für welche man sich entscheidet, kann anschließend entweder *nur* Transition *b* oder aber *nur* Transition *c* schalten. Im Gegensatz dazu erlaubt der entsprechende Interaktionsgraph nach dem Durchlaufen der Aktion *a sowohl* das Durchlaufen von *b als auch* das Durchlaufen von *c*, weil beim Durchlaufen von *a* noch nicht endgültig entschieden wird, um welches *a* es sich hierbei handelt.

6.5.1.3 Hilfstransitionen

Interaktionsgraphen mit Wiederholungen lassen sich prinzipiell durch Einfügen unbenannter *Hilfstransitionen*, deren Schaltvorgang in einer Schalt- bzw. Etikettenfolge ignoriert wird, in äquivalente Petrinetze transformieren (vgl. Abb. 6.12). In ähnlicher Weise kann auch eine Sowohl-als-auch-Verzweigung in ein nahezu äquivalentes Netz umgeformt werden (vgl. Abb. 6.13). Da Petrinetze aber − im Gegensatz zu Interaktionsgraphen − zumeist *echte Parallelität*, d. h. die *gleichzeitige* Ausführung mehrerer Transitionen, erlauben, sind die beiden Spezifikationen in Abb. 6.13 nicht exakt gleichbedeutend. Um die gleichzeitige Ausführung mehrerer Aktionen zu unterbinden, müßte man Petrinetze



Abbildung 6.12: Transformation einer Wiederholung

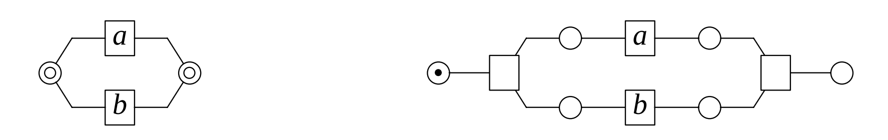


Abbildung 6.13: Prinzipielle Transformation einer Sowohl-als-auch-Verzweigung

[13] Genauer: die *Etikettensprache* des Netzes [Baumgarten96], d. h. die Menge aller Worte $\langle \lambda(t_1), \ldots, \lambda(t_n)\rangle \in \Sigma^*$, für die $\langle t_1, \ldots, t_n\rangle \in T^*$ eine Schaltfolge des Netzes darstellt.

entweder entsprechend uminterpretieren oder aber eine zusätzliche „Synchronisationsstelle“ einführen, die sich im Vor- und Nachbereich jeder Transition des Netzes befindet (vgl. Abb. 6.14).

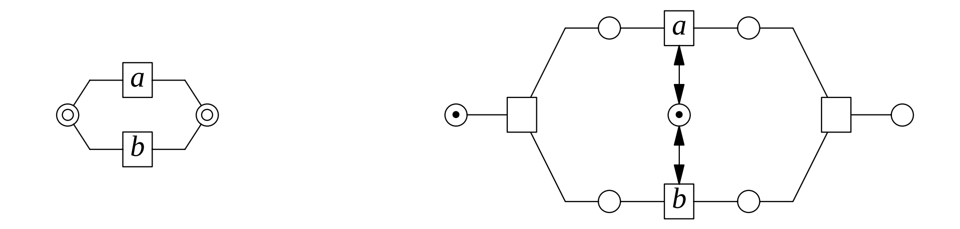


Abbildung 6.14: Exakte Transformation einer Sowohl-als-auch-Verzweigung

6.5.1.4 Kopplung

Noch etwas unangenehmer wird die Situation, wenn man versucht, Interaktionsgraphen mit Kopplungen zu transformieren. Um die Regel umzusetzen, daß Aktionen, die mehreren Zweigen der Kopplung gemeinsam sind, auch gemeinsam durchlaufen werden müssen, kann man versuchen, die betroffenen Aktionen jeweils zu einer einzigen Aktion zu verschmelzen (vgl. Abb. 6.15). Sobald derartige Aktionen jedoch *mehrfach* in einem Zweig auftreten, benötigt man entweder weitere Hilfsstellen und -transitionen (vgl. Abb. 6.16), da prinzipiell jede Traversierung der Aktion *a* im oberen Zweig der Kopplung mit jeder Traversierung von *a* im unteren Zweig synchron erfolgen kann, oder man benötigt insgesamt $m \cdot n$ geeignet zu plazierende Ausprägungen der gemeinsamen Aktion *a*, wenn diese im

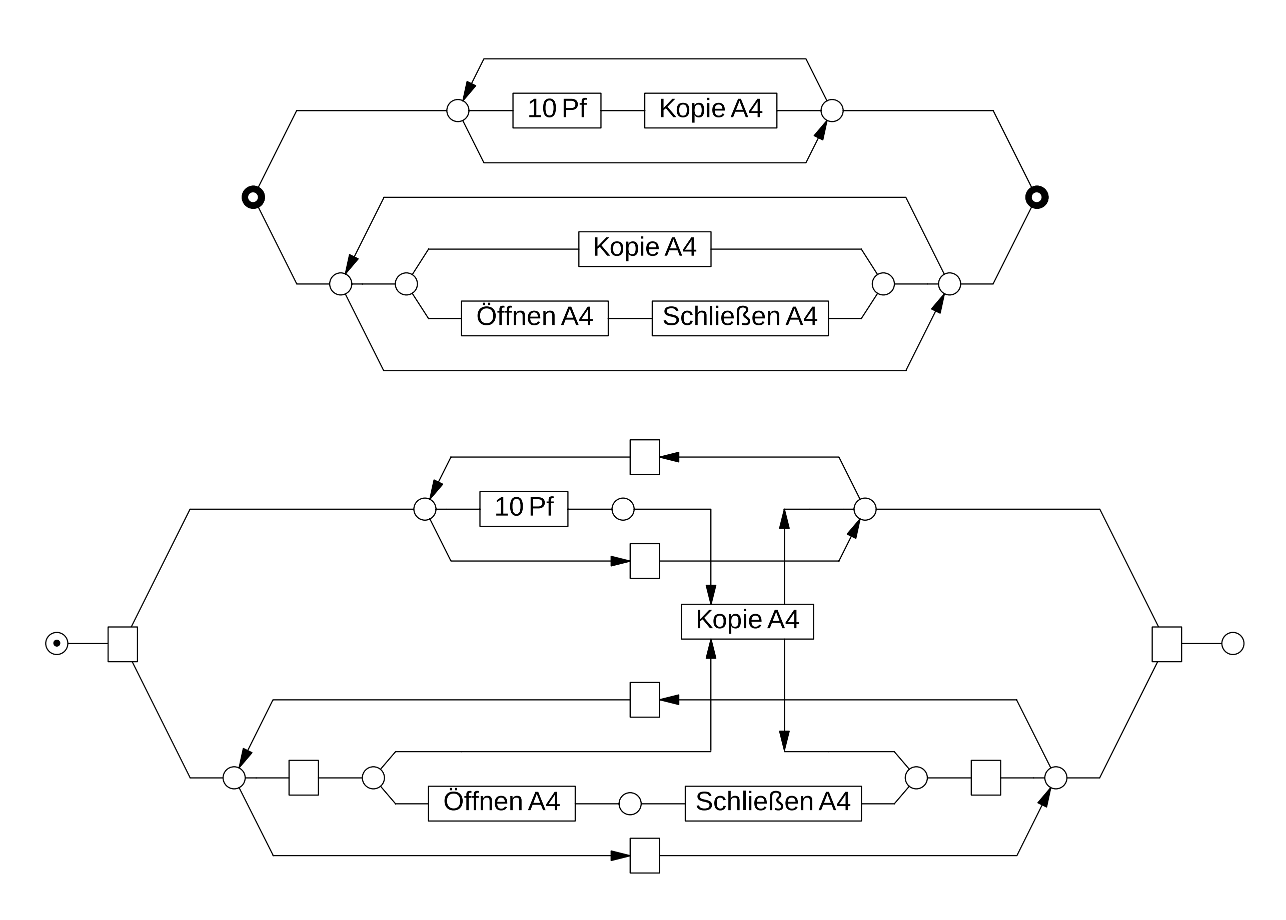


Abbildung 6.15: Transformation einer Kopplung

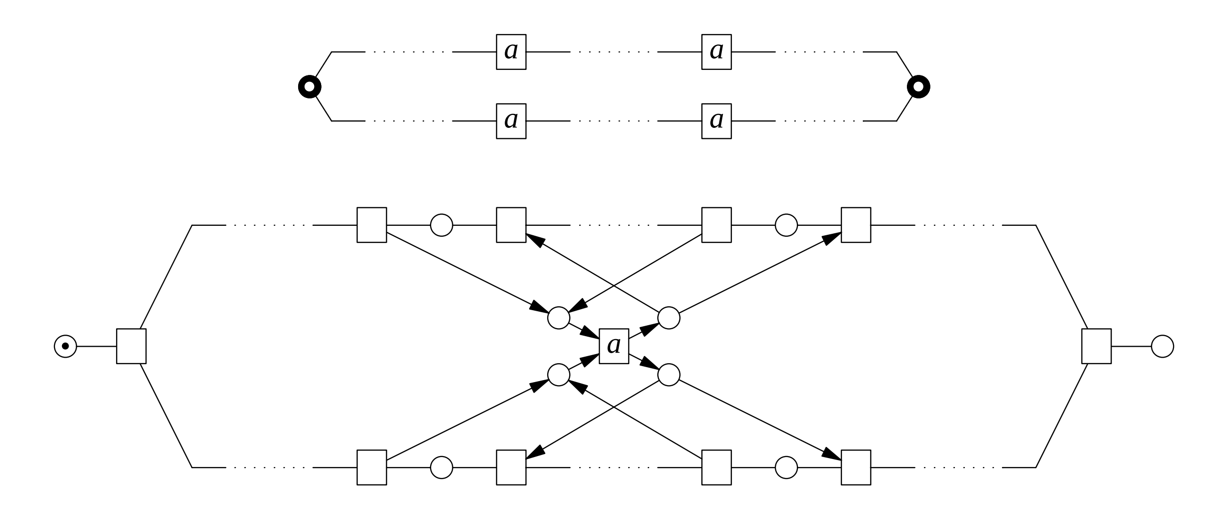


Abbildung 6.16: Verallgemeinerte Transformation von Kopplungen

oberen Zweig *m*-mal und im unteren Zweig *n*-mal auftritt. Für die übrigen gemeinsamen Aktionen der beiden Zweige muß entsprechend verfahren werden.

6.5.1.5 Beliebig-oft-Verzweigung

Sobald man versucht, Interaktionsgraphen mit Beliebig-oft-Verzweigungen auf Petrinetze abzubilden, stößt man mit gewöhnlichen Stellen-Transitionen-Netzen an Grenzen. Um nämlich gewährleisten zu können, daß Transitionen, die in einer Sequenz *hinter* einer Beliebig-oft-Verzweigung liegen, erst schalten dürfen, wenn alle Marken, die in den Rumpf der Verzweigung hineingegangen sind, auch wieder herausgekommen sind, müßte man überprüfen können, ob der Rumpf *markenfrei* ist. Sofern die Markenzahl einer Stelle nicht limitiert ist, gelingt ein solcher Test jedoch nur unter Zuhilfenahme von *Verbotskanten*, die spezifizieren, daß eine Transition schalten darf, wenn eine bestimmte Stelle *keine* Marke trägt [Baumgarten96].[14]

Abbildung 6.17 zeigt eine mögliche Transformationsvorschrift, in der die Verbotskante von s_6 nach t_3 durch einen doppelt durchgestrichenen Pfeil gekennzeichnet ist.[15] Das Teilnetz $s_1 - t_1 - s_2$ entspricht dem linken ◎-Knoten der Beliebig-oft-Verzweigung und wirkt als „Markengenerator", der die Stelle s_2 mit beliebig vielen Marken versorgt, deren Anzahl in der Stelle s_6 „gezählt" wird. Analog entspricht das Teilnetz $s_3 - t_2 - s_4$ dem rechten ◎-Knoten und dient als „Markenverbraucher", der durch das Schalten der Transition t_4 aktiviert wird und anschließend jede Marke, die von t_1 in die Stelle s_2 gelegt wurde und nach Passieren des Verzweigungsrumpfes in der Stelle s_3 erscheint, verbraucht; hierbei wird der „Zähler" s_6 entsprechend dekrementiert. Sobald dieser Zähler den Wert null erreicht, d. h. s_6 keine Marken mehr trägt, kann Transition t_3 schalten und damit die Ausführung der Beliebig-oft-Verzweigung beenden.

Allerdings funktioniert die so konstruierte „Schaltung" nur korrekt, solange Beliebig-oft-Verzweigungen *nicht verschachtelt* werden. Andernfalls tritt das Problem auf, daß die Transition t_3 einer *inneren* Verzweigung nicht unterscheiden kann, ob alle Marken in der Stelle s_6 von einer einzigen oder von mehreren verschiedenen Marken der Stelle s_1 „abstammen", d. h. ob die innere Verzweigung ein- oder mehrmals durchlaufen wird. Im ersten Fall darf t_3 tatsächlich erst schalten, wenn die Stelle s_6 leer ist,

[14] Ist die Markenzahl einer Stelle s höchstens $n \in \mathbb{N}$, so kann man eine zu ihr komplementäre Stelle s' verwalten, so daß die Summe der Marken von s und s' stets n beträgt. In diesem Fall ist die Bedingung „s enthält keine Marke" äquivalent zu der Bedingung „s' enthält n Marken". Eine derartige Limitierung von Markenzahlen ist aber gerade im Kontext von Beliebig-oft-Verzweigungen nicht möglich, da der Rumpf einer solchen Verzweigung *beliebig oft* parallel durchlaufen werden darf.

[15] Bei den Transitionen t_1 bis t_4 handelt es sich um *unbeschriftete* Hilfstransitionen, deren Benennungen lediglich für die nachfolgenden Erläuterungen benötigt werden.

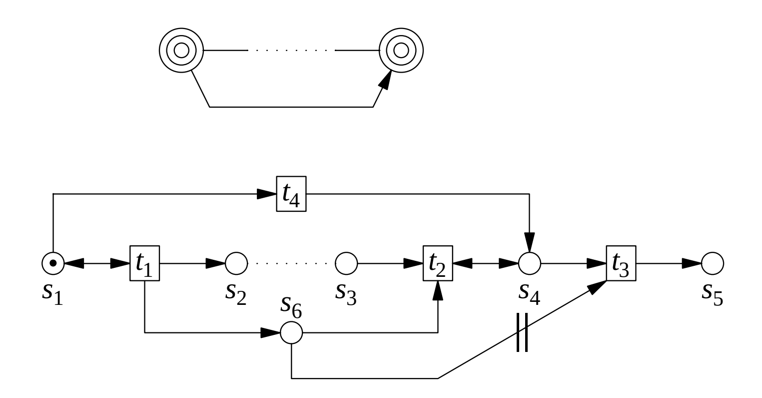


Abbildung 6.17: Transformation von Beliebig-oft-Verzweigungen

im zweiten Fall dürfte sie jedoch bereits schalten, wenn alle Marken einer bestimmen *Sorte* (d. h. alle, die von einer bestimmten Marke der Stelle s_1 abstammen) verbraucht sind.

Zur Veranschaulichung dieser Überlegungen betrachte man den Interaktionsgraphen und das zugehörige Petrinetz in Abb. 6.18. Der Graph kann z. B. durchlaufen werden, indem am äußeren $\circledcirc$-Verzweigungsknoten zwei Gruppen G_1 und G_2 den Rumpf dieser Verzweigung, d. h. den Teilgraphen $a - \circledcirc (b - c) - d$ betreten. Eine dieser Gruppen, beispielsweise G_1, passiert die Aktion a, betritt geschlossen den Rumpf der inneren Beliebig-oft-Verzweigung und durchläuft dort die Aktion b. Die zweite Gruppe G_2 passiert ebenfalls die Aktion a, umgeht dann aber die innere Beliebig-oft-Verzweigung und durchläuft sofort die Aktion d. Die resultierende Ausführungsreihenfolge lautet somit $\langle a, b, a, d \rangle$, d. h. die Aktion d ist zulässig, obwohl sich die Gruppe G_1 noch im Rumpf der inneren Beliebig-oft-Verzweigung befindet.

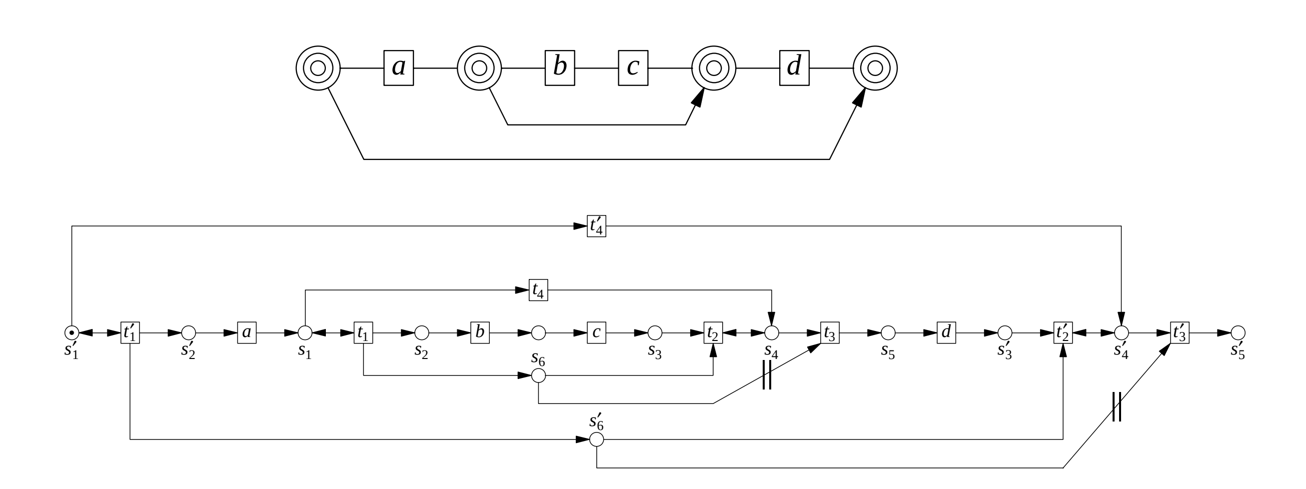


Abbildung 6.18: Verschachtelte Beliebig-oft-Verzweigungen

Diese Ausführungsreihenfolge, die sich natürlich auch mit Hilfe der formalen Semantik von Interaktionsausdrücken herleiten läßt, kann mit dem Petrinetz in Abb. 6.18 jedoch nicht erzielt werden: Zunächst muß die Hilfstransition t'_1 mindestens einmal schalten, bevor die Transition a schalten kann. Unabhängig davon, wie oft t'_1 tatsächlich schaltet, befindet sich nach dem Schalten von a in der Stelle s_1 genau eine Marke. Damit nun als nächstes Transition b schalten kann, muß zuvor die Hilfstransi-

tion t_1 schalten, was zur Folge hat, daß die Stelle s_6 (mindestens) eine Marke enthält, die anzeigt, daß sich eine Gruppe im Rumpf der inneren Beliebig-oft-Verzweigung befindet. Schaltet t_1' erneut, so kann auch a ein zweites Mal schalten, was zur Folge hat, daß s_1 nun zwei Marken enthält, von denen eine sozusagen der Gruppe G_1 und die andere der Gruppe G_2 entspricht. Durch Schalten der Hilfstransition t_4 könnte eine dieser Marken zwar zur Stelle s_4 transferiert werden, was die Hilfstransition t_3 − deren Schalten eine notwendige Voraussetzung für das Schalten von d ist − aber noch nicht aktiviert: Aufgrund der Verbotskante von s_6 nach t_3 kann diese Transition erst schalten, wenn s_6 leer ist. Daher würde das Netz nach Ausführung der Aktionen a, b und a die Ausführung von d verbieten.

6.5.1.6 Netze mit individuellen Marken

Wie bereits angedeutet, liegt die Ursache für dieses unerwünschte Verhalten darin begründet, daß die Hilfstransition t_3, deren Schalten die innere Beliebig-oft-Verzweigung beendet, nicht erkennen kann, daß die Marken in der Stelle s_6 von zwei verschiedenen Marken der Stelle s_1 abstammen. Um das gewünschte Verhalten zu erreichen, bietet sich daher die Verwendung *höherer Petrinetze* mit *individuellen*, d. h. unterscheidbaren Marken an [Jensen91, Baumgarten96]. Abbildung 6.19 zeigt eine Abb. 6.17 entsprechende „Schablone“, die nun nach Belieben verschachtelt werden kann. Die „Generator-Transition“ t_1 konstruiert aus einer beliebigen Marke x der Stelle s_1 sowie einer lokalen „Zählermarke“ n der Stelle s_7 eine eindeutige Marke (x, n) in der Stelle s_2. Außerdem propagiert sie die Marke x wie bisher in die Stelle s_6 sowie in die Stelle s_1 zurück und erhöht die Zählermarke n in der Stelle s_7 um eins. (Initial enthält diese Stelle eine Marke mit dem Wert 1.) Komplementär dazu entnimmt die „Verbraucher-Transition“ t_2 eine von t_1 generierte Marke (x, n) aus der Stelle s_3 sowie eine zu ihr *passende* Marke x aus den Stellen s_4 und s_6 und legt die Marke x in die Stelle s_4 zurück. Die „Abschluß-Transition“ t_3 schließlich entnimmt diese Marke x aus der Stelle s_4 unter der Voraussetzung, daß die Stelle s_6 keine *derartige* Marke enthält, und propagiert sie in die Stelle s_5.

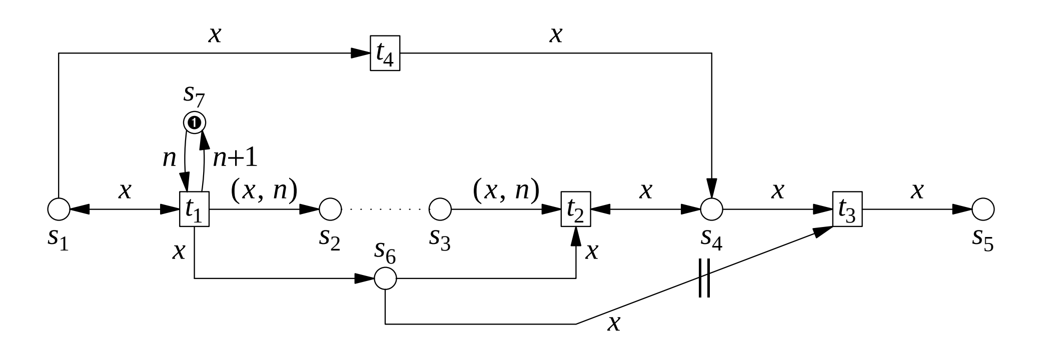


Abbildung 6.19: Allgemeingültige Transformation einer Beliebig-oft-Verzweigung

Durch den Einsatz der Zählermarke n wird gewährleistet, daß die Marken, die in der Stelle s_2 abgelegt werden und anschließend den Rumpf der Beliebig-oft-Verzweigung durchlaufen, *unterscheidbar* sind. Daher erhält die Transition t_1 einer inneren Beliebig-oft-Verzweigung (die sich zwischen s_2 und s_3 der äußeren Verzweigung befindet) unterschiedliche Marken (x, n_1), (x, n_2), ... anstelle von x, die sie in der Stelle s_6 der inneren Verzweigung ablegt. Die Transition t_3 der inneren Verzweigung ist dann schaltbereit, sobald sich in der Stelle s_4 eine Marke (x, n_i) befindet, die Stelle s_6 jedoch keine *derartige* Marke enthält. Daher kann die innere Beliebig-oft-Verzweigung beendet werden, sobald alle Marken, die von *derselben* Marke (x, n_i) abstammen, den Rumpf dieser Verzweigung durchlaufen haben. Daß die Stelle s_6 möglicherweise andere Marken (x, n_j) enthält, die von einem anderen Schaltvorgang der äußeren Transition t_1 stammen und die anzeigen, daß sich noch Marken $(x, n_j, \ldots)$ im Rumpf der inneren Verzweigung befinden, spielt hierbei − wie gewünscht − keine Rolle.

Da eine Eingangsmarke x durch jede Transition t_1, die sie durchläuft, um einen weiteren Zähler n erweitert wird, ist das Verfahren offensichtlich auch für beliebig tief verschachtelte Beliebig-oft-Verzweigungen geeignet.

Anmerkung: Da Stellen-Transitionen-Netze mit Verbotskanten Turing-äquivalent sind [Peterson77], ist die Verwendung individueller Marken offenbar nicht zwingend erforderlich. Allerdings dürfte eine Lösung des Problems mit nicht unterscheidbaren Marken vermutlich noch um einiges komplizierter ausfallen.

6.5.1.7 Parametrisierte Ausdrücke und Quantoren

Durch die Verwendung individueller Marken kann man prinzipiell auch *parametrisierte Aktionen* modellieren, indem man die beim Schalten einer Transition bewegten Marken geeignet als Parameterwerte der zugehörigen Aktion interpretiert. Schließlich lassen sich vermutlich auch Für-ein- und Für-alle-Verzweigungen mit ähnlichen Ideen wie in Abb. 6.19 auf Petrinetze abbilden, sofern man zur Vereinfachung annimmt, daß $\Omega = \mathbb{N}$ gilt, d. h. daß nur natürliche Zahlen als Parameterwerte auftreten.[16] Bei der Transformation einer Für-ein-Verzweigung $\underset{p}{\bigcirc}\, y$ muß man allerdings ein Netz konstruieren, das *zufällig* einen bestimmten Wert für den Parameter p erzeugt, da man am Anfang der Verzweigung noch nicht weiß, welcher Wert tatsächlich beim Durchlaufen des Rumpfs y benötigt wird.

6.5.1.8 Anmerkung

Auch für Pfad- und Flußausdrücke wurden Abbildungen auf Petrinetze vorgenommen und untersucht [Lauer75, Araki81b], wobei die grundsätzliche Transformationsstrategie jeweils sehr ähnlich zu der hier beschriebenen ist. Bei der Abbildung von Pfadausdrücken wurde die „interessante“ parallele Iteration allerdings vollkommen ausgespart, während sie bei der Transformation von Flußausdrücken in keiner anderen Iteration (egal ob parallel oder sequentiell) *enthalten* (d. h. verschachtelt) sein darf. Auf den ersten Blick überraschend ist, daß diese Transformation *ohne Verbotskanten* auskommt, weil sie a priori nur Schaltfolgen betrachtet, an deren Ende alle Stellen außer einer *keine* Marken enthalten. Somit entspricht die erzeugte Petrinetz-*Sprache* zwar der Sprache des Ausdrucks, eine *operationale* Interpretation des Netzes ist jedoch nicht mehr sinnvoll, da Transitionen, die der parallelen Iteration folgen, u. U. bereits schalten können, wenn sich noch Marken im Rumpf der Iteration befinden.[17]

6.5.2 Vergleich von Petrinetzen mit Interaktionsgraphen

Obwohl die vorangegangenen Ausführungen den Eindruck vermitteln könnten, daß sich Interaktionsgraphen prinzipiell auf Petrinetze abbilden lassen, sollten hierbei die folgenden Randbedingungen nicht übersehen werden:

1. Grundsätzlich begibt man sich bei einer solchen Transformation in das bereits mehrfach diskutierte Spannungsfeld von kontrollierter Mehrdeutigkeit gegenüber echtem Nichtdeterminismus: Sofern man nur an der durch ein Petrinetz definierten *Sprache* interessiert ist, handelt es sich bei den vorgestellten Transformationsregeln tatsächlich um *Äquivalenztransformationen*. Versucht man jedoch, die resultierenden Netze nach den üblichen Schaltregeln *auszuführen*, so ist man in Konfliktsituationen gezwungen, *nichtdeterministische* Entscheidungen zu treffen, da das *Aufschieben* von Entscheidungen in der operationalen Semantik von Petrinetzen nicht vorgesehen ist.
2. Das für Interaktionsgraphen wesentliche Prinzip der *modularen Kombination* von Teilgraphen läßt sich mit Petrinetzen nicht verwirklichen, da man bei der Transformation einer *Kopplung* gezwun-

[16] Andernfalls besteht das Problem, die (potentiell unendlich vielen) konkreten Werte $\omega \in \Omega$ durch eine „Generator-Transition“ am Anfang einer Für-alle-Verzweigung zu erzeugen.

[17] Da diese Marken dann auch im Rumpf der Iteration *verbleiben*, wird die gesamte Schaltfolge später verworfen, weil sie nicht die geforderte *Endmarkierung* produziert.

gen ist, die ursprünglichen Teilgraphen bzw. Netze zum Teil erheblich zu modifizieren (vgl. Abb. 6.15 und 6.16).

3. Während sich einfache IAA-Operatoren, wie z. B. Disjunktion oder sequentielle und parallele Komposition, direkt und einfach abbilden lassen, erweist sich die allgemeingültige Transformation der parallelen Iteration (und entsprechend auch die paralleler Quantoren) als so schwierig, daß sie einem ungeübten Anwender auf keinen Fall zugemutet werden kann. Im Gegensatz zu Interaktionsgraphen, ist es bei Petrinetzen auch nicht ohne weiteres möglich, ein solches immer wiederkehrendes syntaktisches Muster in einer *Schablone* zu verbergen (siehe unten).

4. Petrinetze erlauben zwar die Vergröberung bzw. Verfeinerung einzelner Stellen oder Transitionen (d. h. *Abkürzungen* in IAA-Terminologie), nicht jedoch die Verwendung von parametrisierten *Schablonen*, deren *Parameter* ganze Teilgraphen bzw. -netze repräsentieren. Dies bedeutet auch, daß man Petrinetze *nicht* um „benutzerdefinierte Operatoren“ erweitern kann.

Vergleicht man diese Punkte mit den in § 2.8.2 zusammengefaßten Grundprinzipien von Interaktionsausdrücken, so erkennt man, daß nahezu keines dieser Prinzipien von Petrinetzen zufriedenstellend unterstützt wird. Aus diesem Grund wurde der in einer frühen Phase dieser Arbeit unternommene Versuch, Interaktionsgraphen bzw. -ausdrücke auf Petrinetze abzubilden und auf diese Weise sowohl die Semantik als auch die Implementierung des Formalismus sozusagen „umsonst“ zu bekommen, wieder eingestellt.

Anmerkung: Die zuletzt getroffene Aussage sollte nicht etwa dahingehend mißverstanden werden, daß Petrinetze grundsätzlich „schlechter“ als Interaktionsgraphen seien. Es sollte lediglich − ähnlich wie beim Vergleich von Interaktionsausdrücken mit Prozeßalgebren − herausgestellt werden, daß einige, im Rahmen der vorliegenden Arbeit als wesentlich erachtete Prinzipien, von Petrinetzen nicht ausreichend unterstützt werden und sie daher zur Beschreibung von Inter-Workflow-Abhängigkeiten nicht geeignet sind. Die Tatsache, daß Petrinetze an sich − für entsprechende Problemstellungen − ein sehr mächtiges und vielfach bewährtes Hilfsmittel zur Modellierung und Analyse nebenläufiger Systeme darstellen, bleibt hiervon unberührt.

6.6 Workflow-Management-Konzepte

Da Interaktionsausdrücke primär als Formalismus zur Spezifikation und Implementierung von *Inter*-Workflow-Abhängigkeiten konzipiert wurden, soll zum Abschluß dieses Kapitels zum einen auf ihre Beziehung zu Workflow-Beschreibungssprachen − als Formalismus zur Spezifikation von *Intra*-Workflow-Abhängigkeiten − eingegangen werden (§ 6.6.1). Zum anderen soll deutlich gemacht werden, daß Inter-Workflow-Abhängigkeiten trotz ihrer praktischen Relevanz bisher weder in Forschungsprojekten noch in kommerziellen Workflow-Management-Systemen angemessene Beachtung gefunden haben (§ 6.6.2), d. h. daß die vorliegende Arbeit in dieser Beziehung einen wichtigen wissenschaftlichen Beitrag leistet.

6.6.1 Verhältnis von Workflow-Beschreibungssprachen und Interaktionsausdrücken

6.6.1.1 Unterschiede präskriptiver und deskriptiver Formalismen

Auf den ersten Blick mögen die IAA-Operatoren für sequentielle Komposition, Disjunktion und sequentielle Iteration an die für imperative Programmier- oder auch Workflow-Beschreibungssprachen typischen Konstrukte *Sequenz*, *Verzweigung* und *Wiederholung* erinnern. Bei genauerer Betrachtung ergibt sich jedoch ein wesentlicher Unterschied: Während man beispielsweise bei einer If-then-else-Verzweigung oder bei einer Repeat-until-Schleife *explizit* formulieren muß, unter welcher Bedingung der eine oder der andere Zweig durchlaufen bzw. der Schleifenkörper wiederholt wird, werden derarti-

ge Entscheidungen bei Interaktionsausdrücken *implizit* dadurch getroffen, welche Aktionen in der realen Welt ausgeführt werden. Ein imperatives Programm legt also, abhängig von seiner Eingabe oder gewissen externen Bedingungen, exakt fest, welche seiner Anweisungen in welcher Reihenfolge ausgeführt werden, und kann somit als *präskriptive* (*vor*schreibende) Spezifikation bezeichnet werden. Demgegenüber beschreibt ein Interaktionsausdruck in der Regel eine große *Menge* „gleichberechtigter“ Ausführungsreihenfolgen, ohne die tatsächlich auszuführende Folge exakt festzulegen. Dementsprechend kann man Interaktionsausdrücke als *deskriptiven* (*be*schreibenden) Formalismus bezeichnen (vgl. auch § 1.3.4.1).

6.6.1.2 Gegenseitige Ergänzung präskriptiver und deskriptiver Formalismen

Aufgrund dieser prinzipiellen Verschiedenheit werden Interaktionsausdrücke typischerweise für andere Aufgaben oder Problemstellungen eingesetzt als imperative Formalismen. Beispielsweise eignen sie sich sehr gut zur Spezifikation allgemeiner *Integritätsbedingungen* (vgl. z. B. § 5.4.3), deren Formulierung mit imperativen Konstrukten meist sehr mühsam, wenn überhaupt möglich ist. Auf der anderen Seite erweisen sie sich als ungeeignet, wenn es darum geht, *konkrete* Ausführungsreihenfolgen festzulegen, die auf expliziten *Bedingungen* beruhen. Aus diesen Gründen sollten Interaktionsausdrücke und imperative Konstrukte nicht etwa als *Konkurrenten* betrachtet werden, deren Vor- und Nachteile gegeneinander abgewogen werden müßten, sondern vielmehr als *Partner*, die sich gegenseitig ergänzen können, wie dies beispielsweise beim Einsatz von Interaktionsausdrücken als Synchronisationsmechanismus in parallelen Programmiersprachen der Fall ist (vgl. § 5.3.2.2).

Da *Workflow-Beschreibungssprachen* − was den Aspekt der Kontrollflußmodellierung anbelangt − im wesentlichen nichts anderes als graphische Notationen für imperative Konstrukte darstellen, gelten diese Überlegungen für sie in gleicher Weise: Für eine sinnvolle Spezifikation von Workflow-Geflechten werden i. d. R. *sowohl* Workflow-Beschreibungssprachen − zur präskriptiven Modellierung der einzelnen Workflows − *als auch* Interaktionsausdrücke[18] − zur deskriptiven Beschreibung von Inter-Workflow-Abhängigkeiten − benötigt, und es wäre unsinnig, einen der Formalismen gegen den anderen auszuspielen oder krampfhaft zu versuchen, beide Aspekte von Workflow-Geflechten mit *einem* Formalismus zu spezifizieren.

6.6.2 Unterstützung von Inter-Workflow-Abhängigkeiten

6.6.2.1 Stand der Wissenschaft und Technik

Obwohl das Problem der Inter-Workflow-Abhängigkeiten in vielen Anwendungen von Workflow-Management (nicht nur im medizinischen Bereich) von praktischer Bedeutung ist, wird es in der „Workflow community“ nach wie vor „stiefmütterlich“ behandelt. Beispielsweise findet man weder in entsprechenden Themenheften von Zeitschriften [DEB93, DEB95, DPDB95, DSEJ96, IFE97, JIIS98] noch in Büchern, die den Anspruch erheben, den „State of the art“ widerzuspiegeln [Vossen96, Jablonski97], Beiträge, die sich wirklich mit diesem Thema auseinandersetzen. Auch in Konferenzbänden und Workshop-Proceedings, in denen das Thema Workflow-Management an sich immer mehr an Bedeutung gewinnt, sucht man i. d. R. vergeblich nach einschlägigen Artikeln.

Ebenso betrachten heutige Workflow-Management-Systeme − egal ob kommerzielle Produkte oder Forschungsprototypen − einzelne Workflows im wesentlichen als separate, in sich abgeschlossene Prozesse, die *unabhängig* voneinander ausgeführt werden. In der Regel werden weder Beschreibungsmittel zur Spezifikation noch Laufzeitmechanismen zur Überwachung von workflowübergreifenden Synchronisationsbedingungen angeboten. Nur einige wenige Systeme unterstützen ein rudimentäres *Ereigniskonzept*, dessen Schwierigkeiten und Grenzen jedoch bereits in § 1.3.2.2 erläutert wurden. Hinzu kommt, daß man mit derartigen Ansätzen die in § 5.5.2.1 und § 5.5.3.1 beschriebene und für praktische Anwendungen äußerst wichtige Aktualisierung von Arbeitslisten grundsätzlich nicht be-

[18] oder ein vergleichbarer deskriptiver Formalismus

friedigend unterstützen kann, da Aktivitäten durch Ereignisse nur *aktiviert*, nicht jedoch *deaktiviert*, d. h. vorübergehend aus Arbeitslisten entfernt werden können.

Auch in den Aktivitäten und Standardisierungsbemühungen der Workflow Management Coalition (WfMC) wurden Inter-Workflow-Abhängigkeiten bisher nicht berücksichtigt. So enthält das von der Koalition vorgeschlagene Referenzmodell (vgl. § 1.1.6) zwar eine Schnittstelle (Nummer 1), über die einzelne Workflow-Beschreibungen, nicht jedoch Inter-Workflow-Abhängigkeiten ausgetauscht werden können. Auch die Schnittstelle 4, über die ein in Ausführung befindlicher Workflow von einer Ausführungseinheit zur anderen migriert werden kann, erlaubt keine *workflowübergreifende* Synchronisierung von Workflow-Ausführungen. Die Schnittstelle 5 schließlich erlaubt externen Programmen (wie z. B. einem Interaktionsmanager) zwar, den aktuellen Status in Ausführung befindlicher Workflows abzufragen und so prinzipiell die Ausführung von Aktivitäten zu *verfolgen*, sie bietet jedoch keine Möglichkeit, diese Ausführung zu *kontrollieren*, d. h. bei Bedarf zu unterbinden (vgl. § 5.2.1).

6.6.2.2 Semantische Integration von Workflows

Eine der wenigen Arbeiten, die sich explizit mit *interagierenden Workflows* befaßt, ist [Casati96]. Die dort identifizierten Inter-Workflow-Abhängigkeiten sind z. T. recht ähnlich zu denen, die mit Interaktionsausdrücken beschrieben werden können. So entspricht eine *condition-action dependency* $\{ A_1, A_2, \ldots \} <_{ca} B$ beispielsweise dem Ausdruck $\big(A_1 \bigcirc A_2 \bigcirc \ldots\big) - B$.

Allerdings dient der beschriebene Ansatz zur *semantischen Integration von Workflows* nur dazu, die Beschreibungen mehrerer interagierender Workflows in einem „umschließenden Rahmen“ *konzeptuell* zusammenzufassen, um so die Interaktionen zwischen den Workflows besser nachvollziehen zu können. Über eine implementierungstechnische Umsetzung der Inter-Workflow-Abhängigkeiten werden jedoch keine Aussagen gemacht.

Außerdem beschränken sich die Ausführungen auf *kooperierende* Workflows, deren Interaktionen explizit beabsichtigt sind, während *konkurrierende* Workflows, deren Interaktionen eher unerwünscht sind, nicht weiter betrachtet werden. Mehrere überlappende Untersuchungs-Workflows für denselben Patienten sind aber z. B. in diesem Sinne konkurrierend, weil sie alle auf dieselbe „beschränkte Ressource“, nämlich den Patienten, „zugreifen“.

Schließlich ist der beschriebene Integrationsansatz nur für *statische* Workflow-Geflechte anwendbar, für die bereits zur Modellierungszeit sowohl die Anzahl als auch die Typen der interagierenden Workflows genau bekannt sind. Da dies in vielen praktischen Anwendungen jedoch nicht der Fall ist, wurde bei der Entwicklung von Interaktionsausdrücken darauf geachtet, daß sie auch zur Beschreibung *dynamischer* Geflechte verwendet werden können, deren genaue Zusammensetzung sich erst zur Laufzeit ergibt und sich möglicherweise auch dann noch dynamisch verändern kann.

6.6.2.3 Koordinierte Ausführung von Workflows

Im Gegensatz zu [Casati96], wurde im Forschungsprojekt CREW (Correct and Reliable Execution of Workflows) an der University of Massachusetts nicht nur ein Ansatz zur Beschreibung, sondern auch zur implementierungstechnischen Umsetzung von Inter-Workflow-Abhängigkeiten entwickelt [Kamath98], der einen integralen Bestandteil eines Workflow-Management-Systems darstellt. Anders als beim Einsatz von Interaktionsausdrücken, mit denen man workflowübergreifende Integritätsbedingungen *unabhängig* von einzelnen Workflow-Definitionen spezifizieren kann (und muß), werden Bedingungen zur koordinierten Ausführung von Workflows in CREW immer als Teil einer oder mehrerer Workflow-Definitionen formuliert. Dementsprechend ist auch ihre implementierungstechnische Umsetzung unmittelbar mit der Ausführung der einzelnen Workflows verknüpft, da Workflow-Definitionen einschließlich eventueller Integritätsbedingungen auf eine einheitliche Menge von Event-Condition-Action-Regeln (ECA rules) abgebildet werden.

Vergleicht man den Ansatz mit den in § 1.3 diskutierten Versuchen zur Lösung des Workflow-Koordinations-Problems, so entspricht er konzeptionell der expliziten Synchronisierung abhängiger Workflows (§ 1.3.2) mit den dort bereits erläuterten Nachteilen. Zwar gehen die Ausdrucksmöglichkeiten der angebotenen Sprache LAWS (LAnguage for Workflow Specification) über einfache Ereignisope-

rationen hinaus, im Vergleich zu Interaktionsausdrücken sind sie jedoch sehr eingeschränkt. Beispielsweise kann zwar der wechselseitige Ausschluß zweier Aktivitäten direkt formuliert werden (*A* CONFLICTS_WITH *B*), der wechselseitige Ausschluß zweier Aktivitäten-Sequenzen, wie z. B. $A - B$ und $C - D$, muß jedoch mühsam von Hand programmiert werden:

```
// A und C dürfen nicht gleichzeitig ausgeführt werden.
A CONFLICTS_WITH C;

// Wenn A vor C ausgeführt wurde, muß auch B vor C ausgeführt werden.
IF (A RELATIVE_ORDER C) THEN B RELATIVE_ORDER C;

// Wenn C vor A ausgeführt wurde, muß auch D vor A ausgeführt werden.
IF (C RELATIVE_ORDER A) THEN D RELATIVE_ORDER A;
```

Anders als bei Interaktionsausdrücken, müssen sich die in einer Regel auftretenden Aktivitäten wie *A* oder *B* außerdem immer auf eine bestimmte Workflow-Definition (wie z. B. den Sonographie- oder Endoskopie-Workflow) beziehen. Somit ist es nicht möglich, *allgemeingültige* Integritätsbedingungen für Aktivitäten wie Patient vorbereiten, Patient aufklären usw. zu formulieren, die *unabhängig* von ihrer Verwendung in konkreten Workflow-Definitionen gelten sollen. Wie die Beispiele in § 5.4.3 zeigen, treten derartige Integritätsbedingungen in praktischen Anwendungen jedoch sehr häufig auf, weshalb ihre Formulierung mit Interaktionsausdrücken explizit unterstützt wird.

Neben Ausdrucksmitteln zur Spezifikation von Intra- und Inter-Workflow-Abhängigkeiten bietet die Sprache LAWS auch Möglichkeiten zur Formulierung spezieller Fehlerbehandlungs-Strategien, die es in Interaktionsausdrücken nicht gibt. Beispielsweise beschreibt die Regel

```
ON W.A.COMPENSATE
IF (B CONFLICTS_WITH W.A) AND (B HAPPENED_AFTER W.A) THEN
   COMPENSATE B RE-START B;
```

daß die Aktivität *B* des aktuellen Workflows nach einer Kompensierung der Aktivität *A* des Workflows *W* ebenfalls kompensiert und anschließend erneut gestartet werden soll, sofern die beiden Aktivitäten in Konflikt stehen und *B* nach *A* ausgeführt wurde. (In diesem Fall beruhte die Ausführung von *B* möglicherweise auf Resultaten der Ausführung von *A*, die durch die Kompensierung von *A* ungültig geworden sind.) Die Erweiterung von Interaktionsausdrücken um derartige Konzepte bzw. die Entwicklung eines separaten Formalismus zur Beschreibung von *Inter-Workflow-Ausnahmebehandlungen* könnte einen interessanten Aspekt zukünftiger Arbeiten darstellen (vgl. § 7.2.1.1).

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung der Ergebnisse

7.1.1 Kurzfassung

Im Rahmen dieser Arbeit wurden *Interaktionsausdrücke* und *-graphen* als *deskriptiver Formalismus* zur kompakten, übersichtlichen und modularen *Spezifikation* sowie zur effizienten *Implementierung* von Synchronisationsbedingungen unterschiedlichster Art konzipiert, theoretisch untersucht, praktisch implementiert und prototypisch eingesetzt.

Ausgehend von einer ausführlichen, anwendungsorientierten und anschaulichen Beschreibung des Formalismus (Kapitel 2), wurden sukzessive eine präzise *formale Semantik* (Kapitel 3), eine äquivalente *operationale Semantik* (§ 4.5, Anhang B) sowie eine effiziente *Implementierung* (Kapitel 4, insbesondere § 4.6) entwickelt. Aufbauend auf der formalen bzw. operationalen Semantik wurden zahlreiche *formale Eigenschaften* (§ 3.4) sowie Aussagen zur *Ausdrucksmächtigkeit* (§ 3.5) und *Komplexität* (§ 4.7) des Formalismus bewiesen. Darüber hinaus wurde aufgezeigt, wie Interaktionsausdrücke unter anderem zur Spezifikation von *Inter-Workflow-Abhängigkeiten* eingesetzt werden können (§ 5.4) und wie die spezifizierten Bedingungen mit Hilfe eines oder mehrerer *Interaktionsmanager* (§ 5.2) in eine Workflow-Ausführungsumgebung integriert werden können (§ 5.5). Um die Erstellung von Interaktionsgraphen und ihre Transformation in äquivalente Interaktionsausdrücke zu erleichtern, wurde außerdem ein syntaxgesteuerter graphischer *Editor* entwickelt (Anhang C).

Im einzelnen konnten die im folgenden skizzierten Resultate erzielt werden.

7.1.2 Entwicklung von Interaktionsgraphen (Kapitel 2)

Ausgehend von regulären Ausdrücken als Basisformalismus, konnte durch die Hinzunahme einiger weniger zusätzlicher Operatoren und Konzepte ein Formalismus zur Spezifikation von Synchronisationsbedingungen entwickelt werden, der auf der einen Seite konzeptionell einfach und übersichtlich ist und auf der anderen Seite eine große Flexibilität und Ausdrucksmächtigkeit besitzt. Dementsprechend kann er erfolgreich zur Lösung einer großen Palette unterschiedlichster Synchronisationsprobleme eingesetzt werden. Durch die anschauliche graphische Notation (einschließlich eines zugehörigen Editors) sowie durch den Einsatz von Abstraktionsmechanismen lassen sich auch komplizierte Ausdrücke anwenderfreundlich darstellen.

7.1.3 Formale Semantik und Eigenschaften (Kapitel 3)

Durch eine geeignete Verallgemeinerung des „klassischen“ sprachtheoretischen Ansatzes ist es gelungen, eine formale Semantik für Interaktionsausdrücke zu entwickeln, die das anschauliche Prinzip der Graphtraversierung − auch für Spezialfälle wie Sackgassen und endlose Wege − geeignet präzisiert. Aufbauend auf dieser Semantik konnten zahlreiche intuitiv einleuchtende Eigenschaften von Interaktionsausdrücken auch formal verifiziert werden.

7.1.4 Operationale Semantik, Implementierung und Komplexität (Kapitel 4)

Trotz zum Teil erheblicher Schwierigkeiten, insbesondere im Kontext von Quantoren, ist es gelungen, ein vollständiges operationales Modell zur Ausführung von Interaktionsausdrücken zu entwickeln und seine Korrektheit in bezug auf die formale Semantik nachzuweisen. Unter Zuhilfenahme einer geeigneten Programmiersprache konnte dieses Modell von Zuständen, Zustandsübergängen und Zustands-

prädikaten erfolgreich in eine kompakte und effiziente Implementierung von Interaktionsausdrücken überführt werden.

Obwohl es prinzipiell (wie erwartet) Interaktionsausdrücke gibt, für die die Komplexität des Wort- oder Aktionsproblems − bei Verwendung der vorliegenden Implementierung − exponentiell bzgl. der Länge der Eingabe ist, konnten dennoch einige wichtige Aussagen formuliert und bewiesen werden, mit deren Hilfe für sehr viele praktisch relevante Ausdrücke gezeigt werden kann, daß sie gutartig sind, d. h. mit polynomieller Komplexität verarbeitet werden können.

7.1.5 Praktischer Einsatz (Kapitel 5)

Durch die Definition geeigneter Koordinationsprotokolle konnte die zunächst isoliert entwickelte Implementierung von Interaktionsausdrücken mit einer Schnittstelle umgeben werden, die es anderen Programmen erlaubt, zuvor spezifizierte Integritätsbedingungen bei der Ausführung von Aktivitäten tatsächlich zu berücksichtigen. Anhand zweier alternativer Ansätze zur Implementierung von Workflow-Geflechten − Adaption von Arbeitslistenprogrammen einerseits und Adaption von Workflow-Ausführungseinheiten andererseits − wurde aufgezeigt, wie die genannten Koordinationsprotokolle praktisch angewandt werden können. Um Interaktionsgraphen auch in großen verteilten Systemen sinnvoll und zuverlässig einsetzen zu können, wurden Konzepte zur Partitionierung von Graphen, zum Einsatz mehrerer Interaktionsmanager und zum Wiederanlauf nach Systemausfällen erarbeitet. Desweiteren wurde ein einfaches Vorgehensmodell zur Definition von Workflow-Geflechten entwickelt.

7.2 Ausblick

Obwohl Interaktionsausdrücke und -graphen in der vorliegenden Arbeit bereits sehr ausführlich behandelt wurden, kann weder ihre konzeptionelle Entwicklung noch ihre theoretische Untersuchung oder ihre praktische Umsetzung als vollkommen abgeschlossen bezeichnet werden. Im folgenden werden daher einige konzeptionelle Erweiterungsmöglichkeiten sowie offene Fragen aus Theorie und Praxis genannt.

7.2.1 Weiterführende Konzepte

7.2.1.1 Ausnahmebehandlung

Am Ende von Kapitel 5 (§ 5.5.5) wurde bereits angedeutet, daß Interaktionsausdrücke in der Praxis nicht immer wie ursprünglich geplant ausgeführt werden können, weil beispielsweise durch den vorzeitigen Abbruch oder durch *dynamische Änderungen* eines Workflows [Kuhn95ab, Dadam95, Reichert97b, Reichert98ab] bestimmte Aktivitäten nicht wie erwartet zur Ausführung kommen. Damit in einem solchen Fall nicht die gesamte weitere Ausführung eines Workflow-Geflechts durch einen nicht mehr erfüllbaren Interaktionsausdruck blockiert wird, muß es möglich sein, die Ausführung eines Ausdrucks bei Bedarf kontrolliert abzubrechen und an einer definierten Stelle wiederaufzusetzen. Vergleicht man Interaktionsausdrücke mit der Grammatik eines Parsers (z. B. in einem Compiler), so könnte man eine solche Möglichkeit mit den Error-Recovery-Mechanismen des Parsers vergleichen, die es ihm im Falle eines Syntaxfehlers erlauben, die Abarbeitung bestimmter Regeln abzubrechen und an einer anderen Stelle der Grammatik fortzufahren.

Im Rahmen dieser Arbeit wurden derartige Konzepte bereits ansatzweise verfolgt, indem Interaktionsausdrücke um zwei Operatoren $\leftarrow$ und $\rightarrow$ zur Behandlung von *Ausnahmen* erweitert wurden, die gewisse Ähnlichkeiten mit den *Interrupts* von CSP besitzen (vgl. § 6.3.2.1): Ein Ausdruck $z_1 \leftarrow y \rightarrow z_2$ (bzw. ein entsprechender Graph), bestehend aus einem Rumpf y und zwei optionalen „Flügeln“ z_1 und z_2, erlaubt prinzipiell dieselben Ausführungsreihenfolgen wie der Teilausdruck y. Allerdings kann die Traversierung von y jederzeit *abgebrochen* werden, indem mit der Traversierung von z_1 oder z_2 begonnen wird. Nach dem vollständigen Durchlaufen von z_1 kann dann erneut mit der Traversierung von y begonnen werden, während das Ende von z_2 gleichzeitig das Ende des Gesamt-

ausdrucks $z_1 \leftarrow y \rightarrow z_2$ darstellt. Somit ist es möglich, die Abarbeitung des Ausdrucks y entweder durch einen Rückwärtssprung nach z_1 oder durch einen Vorwärtssprung nach z_2 vorzeitig zu beenden.

Anhand konkreter Anwendungsbeispiele müßte nun untersucht werden, ob sich praktisch relevante Problemstellungen mit den skizzierten Operatoren zufriedenstellend lösen lassen. Sollte dies der Fall sein, müßte ihre Semantik präzise definiert und das operationale Modell sowie die Implementierung von Interaktionsausdrücken entsprechend erweitert werden; außerdem sollte eine intuitive graphische Darstellung der Operatoren gefunden und ihre Komplexität untersucht werden. Sollten sich die Operatoren in der Praxis als unzureichend oder unhandlich erweisen, müßten adäquatere Ausdrucksmittel zur Behandlung von Ausnahmesituationen gefunden werden.

Ein anderer Aspekt von Ausnahmebehandlung betrifft das koordinierte Zurücksetzen, Kompensieren oder Erneut-Ausführen von Aktivitäten mehrerer Workflows, wie es in § 6.6.2.3 erwähnt wurde. Hierfür bietet es sich an, bewährte Ansätze zur Beschreibung und Handhabung von *Intra*-Workflow-Ausnahmesituationen, wie z. B. [Reichert97b, Reichert98ab], geeignet auf *Inter*-Workflow-Ausnahmebehandlungen zu verallgemeinern, wobei geklärt werden müßte, inwieweit sich entsprechende Formalismen und Mechanismen mit den Konzepten von Interaktionsgraphen zu einer umfassenden *Inter-Workflow-Koordinations-Sprache* integrieren lassen.

7.2.1.2 Externe Ereignisse

Bereits in Abschnitt 2.7.4 wurde eine als Stoppuhr visualisierte *Pseudoaktivität* pause dazu verwendet, eine bestimmte Wartezeit zwischen zwei Aktivitäten zu erzwingen. In ähnlicher Weise könnte eine Pseudoaktion alarm (visualisiert als Wecker) − die zu einer bestimmten Uhrzeit automatisch ausgeführt wird, sofern sie gerade zulässig ist − dafür sorgen, daß Aktivitäten nur zu bestimmten Uhrzeiten ausgeführt werden können. Beispielsweise würde der Graph in Abb. 7.1 spezifizieren, daß Patienten nur zwischen 8:00 und 16:30 Uhr zu Untersuchungen abgerufen werden dürfen.

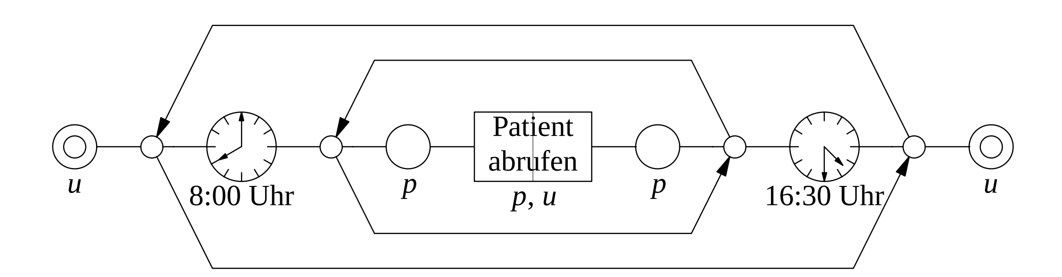


Abbildung 7.1: Anwendung der Pseudoaktion alarm

Prinzipiell kann das Konzept der Pseudoaktionen und -aktivitäten dazu verwendet werden, beliebige *externe Ereignisse* in Interaktionsausdrücke zu integrieren, ohne den Formalismus an sich − einschließlich seiner Semantik und Implementierung − erweitern zu müssen. Für eine konkrete Anwendung müßte jedoch unter anderem geklärt werden, wie Pseudoaktionen syntaktisch von „normalen" Aktionen unterschieden werden können, wie die erforderlichen Parameter (wie z. B. Uhrzeiten, Wartezeiten o. ä.) konkret übergeben werden und wie entsprechende *Agenten*, die für die automatische Ausführung der Pseudoaktionen verantwortlich sind, in eine Gesamtsystemarchitektur integriert werden können.

7.2.2 Theoretische Fragestellungen

7.2.2.1 Transformation und Optimierung von Ausdrücken

Die Liste der formalen Eigenschaften oder Transformationsregeln für Interaktionsausdrücke (§ 3.4) läßt sich sicherlich um weitere nützliche Regeln erweitern, die − wie in Abschnitt 4.7.3.3 bereits angedeutet − dazu verwendet werden können, einen gegebenen Ausdruck x automatisch in einen äquivalenten, aber effizienter implementierbaren Ausdruck x' zu transformieren. Ebenso wie bei der Anfrageoptimierung in Datenbanksystemen [Selinger79, Jarke84], setzt dies jedoch eine möglichst genaue *Kostenschätzung* voraus, die vermutlich nur gelingen kann, wenn man die grobe dreistufige Klassifizierung von Ausdrücken in harmlose, gutartige und bösartige Ausdrücke (§ 4.7.1.3) durch eine feinere Bewertungsskala ersetzt, in der beispielsweise auch der Grad einer polynomiellen Komplexität berücksichtigt wird.

7.2.2.2 Weitere Komplexitätsaussagen

Auch die Liste der positiven Komplexitätsaussagen in § 4.7.5.2 läßt sich sicherlich um weitere Kriterien erweitern, so daß für eine noch größere Menge von Ausdrücken als bisher bewiesen werden kann, daß sie gutartig oder sogar harmlos sind. Eventuell müssen hierfür allerdings weitergehende Optimierungen des Zustandsmodells in Betracht gezogen werden, da die Gutartigkeit oder Harmlosigkeit eines Ausdrucks zum Teil von „unscheinbaren" Zustandsoptimierungen abhängen kann (vgl. § 4.8.1.4).

Außerdem könnte man die Komplexitätsaussagen dahingehend erweitern, daß man nicht nur einzelne Operatoren, sondern auch *Kombinationen* von Operatoren betrachtet, da die Verzweigungsgrade verschachtelter Operatoren oft nicht unabhängig voneinander sind (vgl. § 4.7.5.3). Schließlich könnte es sich lohnen, auch die Struktur der zu verarbeitenden *Worte* in die Komplexitätsbetrachtungen miteinzubeziehen, da sich prinzipiell bösartige Ausdrücke oft nur für bestimmte, gezielt konstruierte und häufig nicht praxisrelevante Aktionsfolgen tatsächlich bösartig verhalten.

7.2.2.3 Ausdrucksmächtigkeit von Interaktionsausdrücken

Für die in § 3.5.3.4 geäußerte Vermutung, daß Interaktionsausdrücke und kontextfreie Grammatiken bzgl. ihrer Ausdrucksmächtigkeit nicht vergleichbar sind, gilt es entweder einen Beweis zu finden, oder aber sie zu widerlegen (d. h. zu zeigen, daß Interaktionsausdrücke ausdrucksstärker als kontextfreie Grammatiken sind). Für letzteres müßte man eine Abbildung angeben, mit der eine beliebige kontextfreie Grammatik in einen äquivalenten Interaktionsausdruck transformiert werden kann, während man für ersteres beweisen müßte, daß sich bestimmte kontextfreie Sprachen nicht mit Hilfe von Interaktionsausdrücken beschreiben lassen.

Vermutlich besitzen „Interaktionssprachen" (d. h. Sprachen, die durch Interaktionsausdrücke definiert werden können) gewisse Abschlußeigenschaften der Art: „Wenn ein bestimmtes Wort w (oder eine Menge von Worten $w_1, \ldots, w_k$) zur Menge $\Psi(x)$ oder $\Phi(x)$ gehört, dann enthält $\Psi(x)$ bzw. $\Phi(x)$ auch das Wort w', das wie folgt aus w (oder $w_1, \ldots, w_k$) konstruiert werden kann: . . ." Eine triviale derartige Eigenschaft ist die Abgeschlossenheit von $\Psi(x)$ bzgl. Präfixbildung (vgl. § 3.4.4). Für reguläre und kontextfreie Sprachen stellt das *Pumping-Lemma* eine solche Eigenschaft dar, und für einen Interaktionsausdruck $x \equiv ⊚\, y$ gilt z. B.:

$$u, v \in \Psi(x) \;\Rightarrow\; u \otimes v \subseteq \Psi(x).$$

Wenn man dann eine kontextfreie Sprache L angeben kann, die zwar das Wort w (oder die Menge der Worte $w_1, \ldots, w_k$), nicht jedoch das Wort w' enthält, hat man auf diese Weise gezeigt, daß L nicht durch einen Interaktionsausdruck beschrieben werden kann.

7.2.2.4 Typisierte Quantorparameter

Für bestimmte Anwendungsgebiete könnte es nützlich sein, *typisierte Quantorparameter* zu verwenden, deren Wertebereich auf eine *Teilmenge* der Menge Ω aller möglichen Werte eingeschränkt ist. Abhängig davon, ob diese Teilmenge endlich oder unendlich ist, kann man dann zwischen endlichen und unendlichen Quantoren unterscheiden. Während sich letztere konzeptionell kaum von den bisher betrachteten Quantoren über der Gesamtmenge Ω unterscheiden (vgl. § 5.2.8.3), müßten endliche Quantoren separat untersucht werden. Beispielsweise würde für sie das Phänomen der endlosen Wege (vgl. § 2.5.2) und damit auch die Notwendigkeit von § 3.4.6 wegfallen. Andererseits könnte man parallele Iterationen *nicht* auf endliche parallele Quantoren zurückführen (vgl. § 3.4.11). Desweiteren würde man für endliche Quantoren tendenziell bessere Komplexitätsaussagen erhalten, d. h. ein Quantorausdruck über einer endlichen Wertemenge könnte gutartig oder harmlos sein, obwohl der entsprechende Ausdruck über einer unendlichen Wertemenge bösartig ist.

7.2.3 Praktische Fragestellungen

7.2.3.1 Integrierte Entwicklungsumgebung für Interaktionsgraphen

Um Interaktionsgraphen noch komfortabler erfassen und pflegen zu können, sollte der prototypisch implementierte Editor (vgl. Anhang C) zu einer *integrierten Entwicklungsumgebung* ausgebaut werden, die es auch erlaubt, Graphen interaktiv zu testen. Durch eine geeignete Kopplung mit der Implementierung von Interaktionsausdrücken könnte der Anwender beispielsweise die Ausführung einzelner Aktionen durch Mausklicks simulieren, wobei ihn das System dadurch unterstützt, daß alle momentan zulässigen Aktionen optisch hervorgehoben werden. Desweiteren wäre eine Kopplung des Editors mit einem zusätzlichen „Komplexitäts- bzw. Optimierer-Modul“ sinnvoll, das den erstellten Graphen bzw. Ausdruck bezüglich seiner Komplexität analysiert und potentiell bösartige Operatoren oder Teilausdrücke optisch kennzeichnet. Eventuell könnte das System sogar Verbesserungsvorschläge anbieten, sofern geeignete Transformationsregeln zur Verfügung stehen. Unabhängig davon könnte ein automatischer Optimierer natürlich auch − wie bei Datenbanksystemen − für den Anwender transparent „unter der Oberfläche“ eingesetzt werden.

7.2.3.2 Integration mit Workflow-Management-Systemen

Obwohl in § 5.5 aufgezeigt wurde, wie sich Workflow-Geflechte durch den Einsatz eines oder mehrerer Interaktionsmanager konkret implementieren lassen, wurde diese Integration von Interaktionsgraphen mit Workflow-Management-Systemen bisher nur rudimentär für ein einziges WfMS tatsächlich durchgeführt. Der Grund für diese „Zurückhaltung“ liegt vor allem darin, daß die in § 5.5.3 beschriebene Adaption von Workflow-Ausführungseinheiten − die gemäß § 5.5.4 der Adaption von Arbeitslistenprogrammen eindeutig überlegen ist − ein sehr aufwendiges Unterfangen darstellt. Erschwerend kommt hinzu, daß heutige Workflow-Management-Systeme für eine derartige Adaption keinerlei Schnittstellen oder Ansatzpunkte vorsehen, da Inter-Workflow-Abhängigkeiten − wie bereits mehrfach erwähnt − bei der Konzeption von Workflow-Management-Systemen bisher überhaupt nicht berücksichtigt wurden.

Nichtsdestotrotz sollten Interaktionsgraphen früher oder später tatsächlich in ein fortschrittliches WfMS integriert werden, da nur so die Voraussetzungen für einen praktischen Einsatz im größeren Stil geschaffen werden können. Da im Rahmen des ADEPT-Projekts an der Universität Ulm [Dadam95, Dadam97, Dadam98] ohnehin ein kompletter Prototyp eines fortschrittlichen WfMSs implementiert wurde, der im Rahmen laufender Forschungsarbeiten kontinuierlich weiterentwickelt wird, bietet es sich an, Interaktionsausdrücke bzw. -graphen konkret in dieses System zu integrieren. Hierbei könnte auch die zuvor erwähnte Interaktionsgraph-Entwicklungsumgebung in die ADEPT-Buildtime-Komponente integriert werden, damit Inter-Workflow-Abhängigkeiten auch rein äußerlich nicht mehr länger ein Anhängsel oder einen Fremdkörper darstellen, sondern ebenso zu der Reihe workflowrelevanter Aspekte gehören wie die Modellierung des Kontroll- und Datenflusses einzelner Work-

flows oder die Beschreibung organisatorischer Strukturen. Es ist die Hoffnung des Verfassers dieser Arbeit, hierfür die notwendigen Grundlagen geschaffen zu haben.

Literaturverzeichnis

[Agrawal92] D. Agrawal, A. El Abbadi: „Transaction Management in Database Systems." In: A. K. Elmagarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992, 1−31.

[Aho75] A. V. Aho, S. C. Johnson, J. D. Ullman: „Deterministic Parsing of Ambiguous Grammars." *Communications of the ACM* 18 (8) August 1975, 441−452.

[Aho79] A. V. Aho, B. W. Kernighan, P. J. Weinberger: „Awk − A Pattern Scanning and Processing Language." *Software-Practice and Experience* 9 (4) April 1979, 267−279.

[Aho88] A. V. Aho, R. Sethi, J. D. Ullman: *Compilerbau* (Teil 1 und 2). Addison-Wesley, Bonn, 1988.

[Alonso95] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, A. El Abbadi, C. Mohan: „Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System." In: S. Laufmann, S. Spaccapietra, T. Yokoi (eds.): *Proc. 3rd Int. Conf. on Cooperative Information Systems (CoopIS)* (Vienna, Austria, May 1995). 1995, 99−110.

[Andler79] S. Andler: „Predicate Path Expressions." In: *Proc. 6th ACM Symp. on Principles of Programming Languages* (San Antonio, TX, January 1979). 1979, 226−236.

[Araki81a] T. Araki, N. Tokura: „Flow Languages Equal Recursively Enumerable Languages." *Acta Informatica* 15, 1981, 209−217.

[Araki81b] T. Araki, T. Kagimasa, N. Tokura: „Relations of Flow Languages to Petri Net Languages." *Theoretical Computer Science* 15, 1981, 51−75.

[Attie93] P. C. Attie, M. P. Singh, A. Sheth, M. Rusinkiewicz: „Specifying and Enforcing Intertask Dependencies." In: R. Agrawal, S. Baker, D. Bell (eds.): *Proc. 19th Int. Conf. on Very Large Data Bases (VLDB)* (Dublin, Ireland, August 1993). 1993, 134−145.

[Baeten90] J. C. M. Baeten, W. P. Weijland: *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, Cambridge, 1990.

[Bal89] H. E. Bal, J. G. Steiner, A. S. Tanenbaum: „Programming Languages for Distributed Computing Systems." *ACM Computing Surveys* 21 (3) September 1989, 261−322.

[Bal95] R. Bal, H. Balsters, R. A. de By, A. Bosschaart, J. Flokstra, M. van Keulen, J. Skowronek, B. Termorshuizen: *The TM Manual* (Version 2.0, Revision E). Technical Report IMPRESS/UT-TECH-T79-001-R2, University of Twente, The Netherlands, June 1995.

[Balsters91] H. Balsters, M. M. Fokkinga: „Subtyping Can Have a Simple Semantics." *Theoretical Computer Science* 87, September 1991, 81−96.

[Balsters93] H. Balsters, R. A. de By, R. Zicari: „Typed Sets as a Basis for Object-Oriented Database Schemas." In: O. M. Nierstrasz (ed.): *ECOOP'93 − Object-Oriented Programming* (7th European Conference; Kaiserslautern, Germany, July 1993; Proceedings). Lecture Notes in Computer Science 707, Springer-Verlag, Berlin, 1993, 161−184.

[Bauer97] T. Bauer, P. Dadam: „A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration.“ In: *2nd IFCIS Int. Conf. on Cooperative Information Systems (CoopIS)* (Kiawah Island, SC, 1997). 1997, 99−108.

[Bauer98] T. Bauer, P. Dadam: *Architekturen für skalierbare Workflow-Management-Systeme. Klassifikation und Analyse*. Nr. 98-02, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, January 1998.

[Bauer99] T. Bauer, P. Dadam: „Verteilungsmodelle für Workflow-Management-Systeme: Klassifikation und Simulation.“ *Informatik Forschung und Entwicklung* 14 (4) December 1999, 203−217.

[Baumgarten96] B. Baumgarten: *Petri-Netze. Grundlagen und Anwendungen* (2. Auflage). Spektrum Akademischer Verlag, Heidelberg, 1996.

[Bergstra90] J. A. Bergstra, J. W. Klop: „An Introduction to Process Algebra.“ In: J. C. M. Baeten (ed.): *Applications of Process Algebra*. Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, Cambridge, 1990, 1−21.

[Bernstein90] P. A. Bernstein, M. Hsu, B. Mann: „Implementing Recoverable Requests Using Queues.“ In: *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 1990, 112−122.

[Berzins77] V. Berzins, D. Kapur: *Denotational and Axiomatic Definitions for Path Expressions*. Computational Structures Group Memo 153-1, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1977.

[Boehm88] H. Boehm, M. Weiser: „Garbage Collection in an Uncooperative Environment.“ *Software-Practice and Experience* 18 (9) September 1988, 807−820.

[Boehm95] H.-J. Boehm, R. Atkinson, M. Plass: „Ropes: An Alternative to Strings.“ *Software-Practice and Experience* 25 (12) December 1995, 1315−1330.

[Bolognesi87] T. Bolognesi, E. Brinksma: „Introduction to the ISO Specification Language LOTOS.“ *Computer Networks and ISDN Systems* 14, 1987, 25−59.

[Brachman91] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick: „Living with CLASSIC: When and How to Use a KL-ONE-Like Language.“ In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 401−456.

[BrinchHansen72] P. Brinch Hansen: „A Comparison of Two Synchronizing Concepts.“ *Acta Informatica* 1 (3) 1972, 190−199.

[Brookes84] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe: „A Theory of Communicating Sequential Processes.“ *Journal of the ACM* 31 (3) July 1984, 560−599.

[Campbell74] R. H. Campbell, A. N. Habermann: „The Specification of Process Synchronization by Path Expressions.“ In: E. Gelenbe, C. Kaiser (eds.): *Operating Systems* (International Symposium; Rocquencourt, France, April 1974; Proceedings). Lecture Notes in Computer Science 16, Springer-Verlag, Berlin, 1974, 89−102.

[Campbell79] R. H. Campbell, R. B. Kolstad: „Path Expressions in Pascal.“ In: *Proc. 4th Int. Conf. on Software Engineering* (Munich, Germany, September 1979). IEEE, 1979, 212−219.

[Campbell80] R. H. Campbell, R. B. Kolstad: „An Overview of Path Pascal's Design" (Including Path Pascal User Manual). *ACM SIGPLAN Notices* 15 (9) September 1980, 13−24.

[Casati96] F. Casati, S. Ceri, B. Pernici, G. Pozzi: „Semantic WorkFlow Interoperability." In: P. Apers, M. Bouzeghoub, G. Gardarin (eds.): *Advances in Database Technology − EDBT'96* (5th Int. Conf. on Extending Database Technology; Avignon, France, March 1996; Proceedings). Lecture Notes in Computer Science 1057, Springer-Verlag, Berlin, 1996, 443−462.

[Chen76] P. P. Chen: „The Entity-Relationship Model − Toward a Unified View of Data." *ACM Transactions on Database Systems* 1 (1) March 1976, 9−36.

[Clark78] K. L. Clark: „Negation as Failure." In: H. Gallaire, J. Minker (eds.): *Logic and Data Bases*. Plenum Press, New York, 1978, 293−322.

[Cleaveland90] R. Cleaveland, J. Parrow, B. Steffen: „The Concurrency Workbench." In: J. Sifakis (ed.): *Automatic Verification Methods for Finite State Systems* (International Workshop; Grenoble, France, June 1989; Proceedings). Lecture Notes in Computer Science 407, Springer-Verlag, Berlin, 1990, 24−37.

[Clocksin94] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[Coleman94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes: *Object-Oriented Development. The Fusion Method*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[Courtois71] P. J. Courtois, F. Heymans, D. L. Parnas: „Concurrent Control with Readers and Writers." *Communications of the ACM* 14 (10) October 1971, 667−668.

[Dadam95] P. Dadam, K. Kuhn, M. Reichert, T. Beuter, M. Nathe: „ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen." In: F. Huber-Wäschle, H. Schauer, P. Widmayer (eds.): *GISI 95. Herausforderungen eines globalen Informationsverbundes für die Informatik* (25. GI-Jahrestagung und 13. Schweizer Informatikertag; Zürich, Switzerland, September 1995). Informatik Aktuell, Springer-Verlag, Berlin, 1995, 677−686.

[Dadam96] P. Dadam: *Verteilte Datenbanken und Client/Server-Systeme. Grundlagen, Konzepte, Realisierungsformen*. Springer-Verlag, Berlin, 1996.

[Dadam97] P. Dadam, W. Klas: „The Database and Information System Research Group at the University of Ulm." *ACM SIGMOD Record* 26 (4) December 1997, 75−79.

[Dadam98] P. Dadam, M. Reichert: „The ADEPT WfMS Project at the University of Ulm." In: *1st European Workshop on Workflow and Process Management (WPM)* (Zürich, Switzerland, 1998). 1998.

[Date98] C. J. Date, H. Darwen: *SQL − Der Standard. SQL/92 mit den Erweiterungen CLI und PSM*. Addison-Wesley, Bonn, 1998.

[DEB93] Special Issue on Workflow and Extended Transaction Systems. *IEEE Data Engineering Bulletin* 16 (2) June 1993.

[DEB95] Special Issue on Workflow Systems. *IEEE Data Engineering Bulletin* 18 (1) March 1995.

[Denert91] E. Denert: *Software-Engineering*. Springer-Verlag, Berlin, 1991.

[Dijkstra68a] E. W. Dijkstra: „Cooperating Sequential Processes.“ In: F. Genuys (ed.): *Programming Languages*. Academic Press, London, 1968.

[Dijkstra68b] E. W. Dijkstra: „The Structure of the T.H.E.-Multiprogramming System.“ *Communications of the ACM* 11 (5) May 1968, 341−346.

[Dijkstra71] E. W. Dijkstra: „Hierarchical Ordering of Sequential Processes.“ *Acta Informatica* 1, 1971, 115−138.

[DPDB95] Special Issue on Software Support for Work Flow Management. *Distributed and Parallel Databases* 3 (2) April 1995.

[DSEJ96] Special Issue on Workflow Systems. *Distributed Systems Engineering Journal* 3 (4) December 1996.

[Elmagarmid92] A. K. Elmagarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Emerson90] E. A. Emerson: „Temporal and Modal Logic.“ In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science* (Volume B: Formal Models and Semantics). Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1990, 995−1072.

[Even95] S. J. Even, F. J. Faase, R. A. de By: *Language Features for Cooperation in an Object-Oriented Database Environment*. Memoranda Informatica 95-40, University of Twente, The Netherlands, September 1995.

[Faase96] F. J. Faase, S. J. Even, R. A. de By: *Introduction to CoCoA* (TransCoop Deliverable IV.3). Technical Report TC/REP/UT/D4-3/033, University of Twente, The Netherlands, February 1996.

[Flanagan98] D. Flanagan: *Java in a Nutshell* (Deutsche Ausgabe der 2. Auflage für Java 1.1). O’Reilly, Cambridge, 1998.

[Frank97] S. Frank, B. Schultheiß: *Prozeßmodellierung und -steuerung mit WorkParty − ein Erfahrungsbericht*. Nr. DBIS-23, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, April 1997.

[Freisleben87] B. Freisleben: *Mechanismen zur Synchronisation paralleler Prozesse*. Informatik-Fachberichte 133, Springer-Verlag, Berlin, 1987.

[Georgakopoulos95] D. Georgakopoulos, M. Hornick, A. Sheth: „An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure.“ *Distributed and Parallel Databases* 3 (2) April 1995, 119−153.

[Gerstner95] R. Gerstner: „Ereignisse als Basis einer Workflow-orientierten Architektur von Anwendungssystemen.“ In: *Proc. GI-Fachtagung MobIS (Modellierung betrieblicher Informationssysteme)* (Bamberg, Germany, October 1995). 1995.

[Govindarajan90] R. Govindarajan, P. Wang, S. Yu: *Statement Tags and Synchronization Expressions in Parallel C (PARC)*. Report No. 260, Dept. of Computer Science, The University of Western Ontario, London, Ontario, Canada, January 1990.

[Govindarajan91] R. Govindarajan, L. Guo, S. Yu, P. Wang: „ParC Project: Practical Constructs for Parallel Programming Languages.“ In: *IEEE Proc. of the 15th Ann. Int. Computer Software and Applications Conference*. 1991, 183−189.

[Gray81] J. Gray: „The Transaction Concept: Virtues and Limitations.“ In: *Proc. 7th Int. Conf. on Very Large Data Bases (VLDB)* (Cannes, France, September 1981). IEEE Computer Society Press, 1981, 144−154.

[Gray93] J. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Guo95] L. Guo: *Synchronization Expressions in Parallel Programming Languages*. Ph. D. Thesis, Dept. of Computer Science, The University of Western Ontario, London, Ontario, Canada, 1995.

[Guo96] L. Guo, K. Salomaa, S. Yu: „On Synchronization Languages.“ *Fundamenta Informaticae* 25 (3+4) March 1996, 423−436.

[Haimowitz96] I. J. Haimowitz, J. Farley, G. S. Fields, J. Stillman, B. Vivier: „Temporal Reasoning for Automated Workflow in Health Care Enterprises.“ In: N. R. Adam, Y. Yesha (eds.): *Electronic Commerce. Current Research Issues and Applications*. Lecture Notes in Computer Science 1028, Springer-Verlag, Berlin, 1996, 87−155.

[Harel87] D. Harel: „Statecharts: A Visual Formalism for Complex Systems.“ *Science of Computer Programming* 8, 1987, 231−274.

[Heinsohn92] J. Heinsohn, D. Kudenko, B. Nebel, H.-J. Profitlich: *An Empirical Analysis of Terminological Representation Systems*. RR-92-16, Deutsches Forschungsinstitut für künstliche Intelligenz (DFKI), Saarbrücken, 1992.

[Hennessy88] M. Hennessy: *Algebraic Theory of Processes*. The MIT Press, Cambridge, MA, 1988.

[Hoare85] C. A. R. Hoare: *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[Hollingsworth94] D. Hollingsworth: *The Workflow Reference Model* (Issue 1.1). Document Number WFMC-TC-1003, Workflow Management Coalition, Brussels, Belgium, November 1994.

[Hopcroft90] J. E. Hopcroft, J. D. Ullman: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, Bonn, 1990.

[Hull87] R. Hull, R. King: „Semantic Database Modeling: Survey, Applications, and Research Issues.“ *ACM Computing Surveys* 19 (3) September 1987, 201−260.

[IFE97] Themenheft Workflow-Management. *Informatik Forschung und Entwicklung* 12 (2) May 1997.

[ISO87] ISO−Information Processing Systems−Open Systems Interconnection: *LOTOS − A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. DIS 8807, ISO, 1987.

[Jablonski95a] S. Jablonski: „Workflow-Management-Systeme: Motivation, Modellierung, Architektur.“ *Informatik-Spektrum* 18 (1) February 1995, 13−24.

[Jablonski95b] S. Jablonski: *Workflow-Management-Systeme: Modellierung und Architektur*. Thomson's Aktuelle Tutorien 9, International Thomson Publishing, Bonn, 1995.

[Jablonski97] S. Jablonski, M. Böhm, W. Schulze (eds.): *Workflow-Management. Entwicklung von Anwendungen und Systemen*. dpunkt-Verlag, Heidelberg, 1997.

[Jarke84] M. Jarke, J. Koch: „Query Optimization in Database Systems.“ *ACM Computing Surveys* 16 (2) June 1984, 111−152.

[Jensen91] K. Jensen, G. Rozenberg (eds.): *High-Level Petri Nets. Theory and Application*. Springer-Verlag, 1991.

[JIIS98] Special Issue on Workflow and Process Management. *Journal of Intelligent Information Systems. Integrating Artificial Intelligence and Database Technologies* 10 (2) March 1998.

[Jones96] R. Jones, R. Lins: *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Chichester, 1996.

[Kamath98] M. Kamath, K. Ramamritham: „Failure Handling and Coordinated Execution of Concurrent Workflows.“ In: *Proc. 14th Int. Conf. on Data Engineering (ICDE)* (Orlando, FL, February 1998). IEEE Computer Society, 1998, 334−341.

[Kernighan86] B. W. Kernighan, R. Pike: *Der UNIX-Werkzeugkasten*. Carl Hanser Verlag, München, 1986.

[Kernighan90] B. W. Kernighan, D. M. Ritchie: *Programmieren in C* (Zweite Ausgabe: ANSI-C). Carl Hanser Verlag, München, 1990.

[Klein91] J. Klein: „Advanced Rule Driven Transaction Management“ (Extended Abstract). In: *Proc. 36th IEEE Computer Society Int. Conf. (COMPCON)* (San Francisco, CA, March 1991). 1991, 562−567.

[Koenig95] A. Koenig, B. Stroustrup: „Foundations for Native C++ Styles.“ *Software-Practice and Experience* 25 (S4) December 1995, 45−86.

[Kofler93] T. Kofler: „Robust Iterators in ET++.“ *Structured Programming* 14, 1993, 62−85.

[Konyen96a] I. Konyen, M. Reichert, B. Schultheiß, R. Mangold: *Ein Prozeßentwurf für den Bereich der minimal invasiven Chirurgie*. Nr. DBIS-14, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, July 1996.

[Konyen96b] I. Konyen, B. Schultheiß, M. Reichert: *Prozeßentwurf für den Ablauf einer radiologischen Untersuchung*. Nr. DBIS-15, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, July 1996.

[Konyen96c] I. Konyen, B. Schultheiß, M. Reichert: *Prozeßentwurf eines Ablaufs im Labor*. Nr. DBIS-16, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, July 1996.

[Kuhn95a] K. Kuhn, M. Reichert, P. Dadam: *Using Workflow Management Systems in Clinical Environments: A Critical Analysis*. Nr. DBIS-2, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, June 1995.

[Kuhn95b] K. Kuhn, M. Reichert, P. Dadam: „Unterstützung der klinischen Kooperation durch Workflow-Management-Systeme: Anforderungen, Probleme, Perspektiven.“ In: H. J. Trampisch, S. Lange (eds.): *Medizinische Forschung − Ärztliches Handeln* (Proc. 40. Jahrestagung der GMDS). MMV Medizin Verlag, München, 1995, 437−441.

[Lauer75] P. E. Lauer, R. H. Campbell: „Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes.“ *Acta Informatica* 5, 1975, 297−332.

[Lauer79] P. E. Lauer, M. W. Shields, E. Best: „COSY − A System Specification Langauge Based on Paths and Processes.“ *Acta Informatica* 12, 1979, 109−158.

[Lawless91] J. A. Lawless, M. M. Miller: *Understanding CLOS. The Common Lisp Object System*. Digital Press, 1991.

[Lutz96] M. Lutz: *Programming Python*. O’Reilly, Cambridge, 1996.

[Lutz99] M. Lutz, D. Ascher: *Learning Python*. O’Reilly, Cambridge, 1999.

[MacGregor91] R. MacGregor: „The Evolving Technology of Classification-Based Knowledge Representation Systems.“ In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 385−400.

[May83] D. May: „Occam.“ *ACM SIGPLAN Notices* 18 (4) April 1983, 69−79.

[Milner80] R. Milner: *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92, Springer-Verlag, Berlin, 1980.

[Milner89] R. Milner: *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[Mohan94] C. Mohan, D. Dievendorff: „Recent Work on Distributed Commit Protocols and Recoverable Messaging and Queuing.“ *IEEE Data Engineering Bulletin* 17 (1) March 1994, 22−28.

[Moller99] F. Moller, P. Stevens: *The Edinburgh Concurrency Workbench User Manual* (Version 7.1). Laboratory for Foundations of Computer Science, The University of Edinburgh, July 1999.

[Moss81] J. E. B. Moss: *Nested Transactions. An Approach to Reliable Distributed Computing*. Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, 1981.

[Nodine92] M. H. Nodine, S. Ramaswamy, S. B. Zdonik: „A Cooperative Transaction Model for Design Databases.“ In: A. K. Elmagarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992, 53−85.

[Nye92a] A. Nye: *Xlib Programming Manual for Version 11 of the X Window System*. O’Reilly, Sebastopol, CA, 1992.

[Nye92b] A. Nye: *Xlib Reference Manual for X11, Release 4 and Release 5*. O’Reilly, Sebastopol, CA, 1992.

[Ogden78] W. F. Ogden, W. E. Riddle, W. C. Rounds: „Complexity of Expressions Allowing Concurrency.“ In: *Proc. 5th ACM Symp. on Principles of Programming Languages*. 1978, 185−194.

[Özsu91] M. T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Peterson77] J. L. Peterson: „Petri Nets.“ *ACM Computing Surveys* 9 (3) September 1977, 223−252.

[Rastogi93] R. Rastogi, S. Mehrotra, H. F. Korth, A. Silberschatz: „Transcending the Serializability Requirement.“ *IEEE Data Engineering Bulletin* 16 (2) June 1993, 8−11.

[Reichert97a] M. Reichert, K. Kuhn, P. Dadam: *Optimierung von Leistungsprozessen im Krankenhaus. Erfahrungen, Perspektiven und Grenzen*. Nr. DBIS-28, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, June 1997.

[Reichert97b] M. Reichert, P. Dadam: „A Framework for Dynamic Changes in Workflow Management Systems.“ In: *8th Int. Workshop on Database and Expert Systems Applications (DEXA)* (Toulouse, France, September 1997). 1997, 42−48.

[Reichert98a] M. Reichert, P. Dadam: „ADEPT/flex: Supporting Dynamic Changes of Workflows Without Losing Control.“ *Journal of Intelligent Information Systems. Integrating Artificial Intelligence and Database Technologies* 10 (2) March 1998, 93−129.

[Reichert98b] M. Reichert, C. Hensinger, P. Dadam: „Supporting Adaptive Workflows in Advanced Application Environments.“ In: *Proc. EDBT Workshop on Workflow Management Systems* (Valencia, Spain, March 1998). 1998, 100−109.

[Reisig86] W. Reisig: *Petrinetze: Eine Einführung*. Studienreihe Informatik, Springer-Verlag, Berlin, 1986.

[Reiter78] R. Reiter: „On Closed World Data Bases.“ In: H. Gallaire, J. Minker (eds.): *Logic and Data Bases*. Plenum Press, New York, 1978, 55−76.

[Reuter87] A. Reuter: „Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen.“ In: P. C. Lockemann, J. W. Schmidt (eds.): *Datenbank-Handbuch*. Springer-Verlag, Berlin, 1987, 337−479.

[Riddle73] W. E. Riddle: „A Method for the Description and Analysis of Complex Software Systems.“ *ACM SIGPLAN Notices* 8 (9) September 1973, 133−136.

[Riddle79a] W. E. Riddle: „An Approach to Software System Behavior Description.“ *Computer Languages* 4, 1979, 29−47.

[Riddle79b] W. E. Riddle: „An Approach to Software System Modelling and Analysis.“ *Computer Languages* 4, 1979, 49−66.

[Rochkind88] M. J. Rochkind: *UNIX-Programmierung für Fortgeschrittene*. Carl Hanser Verlag, München, 1988.

[Rupietta94] W. Rupietta, G. Wernke: „Umsetzung organisatorischer Regelungen in der Vorgangsbearbeitung mit WorkParty und ORM.“ In: U. Hasenkamp, S. Kirn, M. Syring (eds.): *CSCW − Computer Supported Cooperative Work. Informationssysteme für dezentralisierte Unternehmensstrukturen*. Addison-Wesley, Bonn, 1994, 135−154.

[Schildt99] H. Schildt: *STL Programming from the Ground Up*. Osborne/McGraw-Hill, Berkeley, CA, 1999.

[Schöning95] U. Schöning: *Theoretische Informatik − kurzgefaßt* (2. Auflage). Spektrum Akademischer Verlag, Heidelberg, 1995.

[Schultheiß95a] B. Schultheiß, J. Meyer, R. Mangold, T. Zemmler, M. Reichert: *Prozeßentwurf für den Ablauf einer stationären Chemotherapie*. Nr. DBIS-5, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, November 1995.

[Schultheiß95b] B. Schultheiß, J. Meyer, R. Mangold, T. Zemmler, M. Reichert: *Prozeßentwurf am Beispiel eines Ablaufs aus dem OP-Bereich*. Nr. DBIS-6, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, December 1995.

[Schultheiß96] B. Schultheiß, J. Meyer, R. Mangold, T. Zemmler, M. Reichert: *Prozeßentwurf für den Ablauf einer ambulanten Chemotherapie*. Nr. DBIS-7, Interne Ulmer Informatik-Berichte, Abteilung Datenbanken und Informationssysteme, Universität Ulm, January 1996.

[Schulz90] A. Schulz: *Software-Entwurf. Methoden und Werkzeuge* (2. Auflage). R. Oldenbourg Verlag, München, 1990.

[Schütte88] A. Schütte: *Programmieren in Occam*. Addison-Wesley, Bonn, 1988.

[Selinger79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price: „Access Path Selection in a Relational Database Management System." In: *Proc. ACM SIGMOD Int. Conf. on Management of Data* (Boston, MA, June 1979). 1979, 23−34.

[Shaw78] A. C. Shaw: „Software Description with Flow Expressions." *IEEE Transactions on Software Engineering* SE-4 (3) May 1978, 242−254.

[Shaw80a] A. C. Shaw: „On the Specification of Graphics Command Languages and Their Processors." In: R. A. Guedj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce (eds.): *Methodology of Interaction* (IFIP Workshop on Methodology of Interaction; Seillac, France, May 1979). North-Holland Publishing Company, Amsterdam, 1980, 377−392.

[Shaw80b] A. C. Shaw: „Software Specification Languages Based on Regular Expressions." In: W. E. Riddle, R. E. Fairley (eds.): *Software Development Tools*. Springer-Verlag, Berlin, 1980, 148−175.

[SNI95] SNI: *WorkParty Benutzerhandbuch* (Version 2.0). Siemens Nixdorf Informationssysteme AG, August 1995.

[Steele90] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[Stevens92] W. R. Stevens: *Programmieren von Unix-Netzen*. Carl Hanser Verlag, München, 1992.

[Stroustrup83] B. Stroustrup: „Adding Classes to C: An Exercise in Language Evolution." *Software-Practice and Experience* 13, 1983, 139−161.

[Stroustrup95] B. Stroustrup: *Die C++ Programmiersprache* (Erweitert um Entwürfe zur ANSI-/ISO-Standardisierung; 2. Auflage). Addison-Wesley, Bonn, 1995.

[Tang95] J. Tang, J. Veijalainen: „Enforcing Inter-Task Dependencies in Transactional Workflows." In: S. Laufmann, S. Spaccapietra, T. Yokoi (eds.): *Proc. 3rd Int. Conf. on Cooperative Information Systems (CoopIS)* (Vienna, Austria, May 1995). 1995, 72−86.

[Versteegen95] G. Versteegen: „Die Ansätze der Workflow Management Coalition.“ *iX Multiuser Multitasking Magazin* 3/95, March 1995, 152−160.

[Vissers91] C. A. Vissers, G. Scollo, M. van Sinderen, E. Brinksma: „Specification Styles in Distributed Systems Design and Verification.“ *Theoretical Computer Science* 89, 1991, 179−206.

[Vogel92] P. Vogel, R. Erfle: „Backtracking Office Procedures.“ In: A. M. Tjoa, I. Ramos (eds.): *Proc. Int. Conf. on Database and Expert Systems Applications (DEXA)* (Valencia, Spain, 1992). Springer-Verlag, Berlin, 1992, 506−511.

[Vossen96] G. Vossen, J. Becker (eds.): *Geschäftsprozeßmodellierung und Workflow-Management. Modelle, Methoden, Werkzeuge*. International Thomson Publishing, Bonn, 1996.

[Wächter95a] H. Wächter, F. J. Fritz, A. Berthold, B. Drittler, H. Eckert, R. Gerstner, R. Götzinger, R. Krane, A. Schaeff, C. Schlögel, R. Weber: „Modellierung und Ausführung flexibler Geschäftsprozesse mit SAP Business Workflow 3.0.“ In: F. Huber-Wäschle, H. Schauer, P. Widmayer (eds.): *GISI 95. Herausforderungen eines globalen Informationsverbundes für die Informatik* (25. GI-Jahrestagung und 13. Schweizer Informatikertag; Zürich, Switzerland, September 1995). Informatik Aktuell, Springer-Verlag, Berlin, 1995, 197−204.

[Wächter95b] H. Wächter: „Flexible Business Processing with SAP Business Workflow 3.0“ (Invited Presentation; Extended Abstract). In: *Int. Workshop on High Performance Transaction Processing Systems* (Asilomar, CA, September 1995). 1995.

[Weikum92] G. Weikum, H.-J. Schek: „Concepts and Applications of Multilevel Transactions and Open Nested Transactions.“ In: A. K. Elmagarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992, 515−553.

[WfMC96] Members of the WfMC: *Terminology & Glossary* (Issue 2.0). Document Number WFMC-TC-1011, Workflow Management Coalition, Brussels, Belgium, June 1996.

[Winston89] P. H. Winston, B. K. P. Horn: *LISP* (Third Edition). Addison-Wesley, Reading, MA, 1989.

[Wirth82] N. Wirth: *Programming in Modula-2*. Springer-Verlag, 1982.

[Wirth88] N. Wirth: „The Programming Language Oberon.“ *Software-Practice and Experience* 18 (7) July 1988, 671−690.

[Yu96] S. Yu: *E-Mail-Kommunikation über Synchronisierungsausdrücke*. Dept. of Computer Science, The University of Western Ontario, London, Ontario, Canada, September 1996.

[Zave85] P. Zave: „A Distributed Alternative to Finite-State-Machine Specifications.“ *ACM Transactions on Programming Languages and Systems* 7 (1) January 1985, 10−36.

Anhang A

Die Programmiersprache CH

A.1 Motivation

A.1.1 Grenzen imperativer und objektorientierter Programmiersprachen

Die Programmiersprache CH, die zur Implementierung von Interaktionsausdrücken verwendet wurde (vgl. Kapitel 4, insbesondere § 4.3.3, § 4.4.2 und § 4.6), stellt eine persönliche und pragmatische Antwort des Verfassers dieser Arbeit auf eine Reihe von Unzulänglichkeiten herkömmlicher imperativer Programmiersprachen (wie z. B. Modula oder C) dar, die in der alltäglichen Programmierpraxis immer wieder zu „Reibungs- und Effizienzverlusten" führen und auch durch objektorientierte Sprachen (wie z. B. Smalltalk oder Java) nicht oder nur bedingt überwunden werden. Obwohl manche dieser Lücken durch geeignete Bibliotheksfunktionen geschlossen werden können, stößt man bei vielen anderen doch an prinzipielle Grenzen der Ausdrucksmächtigkeit dieser Sprachen, beispielsweise bei dem Versuch, allgemeine *Containertypen* wie Listen oder Mengen zu implementieren, die anschließend *typsicher* für beliebige Elementtypen verwendet werden können.

Um das zuletzt genannte Problem befriedigend lösen zu können, muß *Generizität* (engl. genericity) als Grundkonzept in der Programmiersprache verankert sein, wodurch die Anzahl der geeigneten „Kandidatinnen" bereits drastisch eingeschränkt wird. Aufgrund ihres großen Bekanntheitsgrades und ihrer weiten Verbreitung drängt sich hier die Sprache C++ [Stroustrup95] förmlich auf, die das Konzept der Generizität mit Hilfe von *Template*-Klassen und -Funktionen relativ gut unterstützt.

Auf der anderen Seite wirkt C++ jedoch durch ein Überangebot an Konzepten und durch teilweise fehlerhafte oder inkompatible Implementierungen nicht unbedingt „vertrauenerweckend". Außerdem vermißt man trotz dieses Überangebots nach wie vor einige wichtige Elemente, wie z. B. ein klares *Modulkonzept* mit expliziten Import/Export-Schnittstellen (wie man es z. B. von Modula [Wirth82] oder Oberon [Wirth88] kennt) oder Mechanismen zur *automatischen Speicherbereinigung* (engl. garbage collection).

A.1.2 Offene Typdefinitionen

Ebenso wie in anderen imperativen oder objektorientierten Sprachen, ist man auch in C++ gezwungen, einen benutzerdefinierten Typ (wie z. B. einen Recordtyp oder eine Klasse) immer *zusammen* mit seiner Struktur (d. h. seinen Komponenten oder Attributen) zu definieren, wobei die Definition der Attribute der Definition des Typs hierarchisch *untergeordnet* ist. Demgegenüber stehen Ansätze aus dem Bereich der semantischen Datenmodellierung [Hull87], wie z. B. das bekannte Entity-Relationship-Modell [Chen76], bei dem Entitäten (oder Typen) und Beziehungen (oder Attribute) *gleichberechtigt* nebeneinander stehen und auch nicht notwendigerweise „gleichzeitig" (d. h. an derselben Stelle eines Programms) definiert werden müssen. Ähnliches gilt für Beschreibungstechniken wie z. B. terminologische Logiken [MacGregor91, Brachman91, Heinsohn92] aus dem Bereich der Wissensrepräsentation, bei denen Begriffe (engl. concepts) und Rollen (engl. roles) relativ unabhängig voneinander definiert werden können und somit beispielsweise die Menge der Rollen (d. h. Attribute) eines Begriffs (Typs) auch noch „nachträglich" (d. h. an einer ganz anderen Stelle des Programms) *erweitert* werden kann.

Diese, nach Meinung des Verfassers dieser Arbeit sehr natürliche Vorgehensweise bei der Definition von Datenstrukturen, läßt sich durch das objektorientierte Konzept der „Typerweiterung" durch *Vererbung* bzw. *Subtypbildung* nur bedingt nachbilden, weil eine derartige „Typerweiterung" keine wirkliche *Erweiterung* eines *bestehenden* Typs, sondern vielmehr die Definition eines *neuen* „erweiterten" Typs darstellt. Konsequenterweise existiert das Konzept der Vererbung oder Subtypbildung in

terminologischen Logiken (oder auch erweiterten Entity-Relationship-Modellen) zusätzlich und *orthogonal* zu der beschriebenen Möglichkeit der *inkrementellen* bzw. *offenen Typdefinition*.

A.1.3 C++ als Basissprache

Trotz dieser Mängel bleibt C++ zumindest als *Basissprache* interessant, da sie durch ihren „Multi-Paradigmen-Charakter" − beispielsweise wird klassisches imperatives oder auch funktional ausgerichtetes Programmieren ebenso unterstützt wie typisch objektorientierte Vorgehensweisen − sowie durch eine Vielzahl fortgeschrittener Konzepte (wie z. B. Templates, Ausnahmebehandlung oder Überladen von Operatoren) die Möglichkeit bietet, *Spracherweiterungen* bis zu einem gewissen Grad in der Sprache selbst oder aber mit Hilfe einfacher Präprozessoren (die nur einige wenige Schlüsselworte interpretieren müssen) zu implementieren. So ist man beispielsweise durch automatische Konstruktor- und Destruktoraufrufe prinzipiell in der Lage, automatische Speicherbereinigung auf Bibliotheksebene zu implementieren.

Vor diesem Hintergrund könnte man die Programmiersprache CH als eine mit moderatem Aufwand realisierte *Erweiterung von C++* bezeichnen, die ihre „Muttersprache" C++ unter anderem um *offene Typen* (siehe oben), *automatische Speicherbereinigung, partielle Funktionen* und explizite *Import/Export-Schnittstellen* erweitert (siehe auch § A.2.2). Da auf der anderen Seite jedoch fast alle Konzepte gemieden oder höchstens zur Implementierung der Spracherweiterungen benutzt werden, die man typischerweise mit C++ assoziiert (Klassen, Vererbung, virtuelle Funktionen), ist diese Bezeichnung eher irreführend. Eigentlich handelt es sich um ein *besseres C* [Koenig95] oder um ein *C mit offenen Typen, partiellen Funktionen etc.* anstelle eines *C mit Klassen* [Stroustrup83], wie C++ anfangs genannt wurde. Allerdings sollte die Bezeichnung *besseres C* ebenfalls nicht mit *Erweiterung von C* gleichgesetzt werden, da es in CH (auf Anwenderebene) beispielsweise keine Zeiger gibt! Somit trifft die Bezeichnung *Variante von C bzw. C++* vermutlich am besten die eigentliche Intention.

Anmerkung: Der Buchstabe H des Namens CH entsteht, wenn man die Symbolfolge ++ des Namens C++ horizontal kräftig beschneidet und vertikal etwas dehnt. Dies soll zum Ausdruck bringen, daß die Sprache C++ zunächst erheblich eingeschränkt und anschließend an einigen wenigen Stellen gezielt erweitert wurde. Auch die Verwendung der Schriftart Helvetica Narrow soll darauf hinweisen, daß CH − im Gegensatz zu ihrer „Mutter" C++ − eine schlanke Sprache mit wenigen Kernkonzepten darstellt.

A.2 Sprachumfang

A.2.1 Ausgewählte Konzepte von C++

Neben den meisten Konzepten von ANSI-C [Kernighan90], d. h. einer standardisierten und typsicheren Version von C, enthält die Sprache CH die folgenden *ausgewählten Konzepte von C++*:

- Variablen-Deklarationen können nicht nur am Anfang, sondern auch zwischen den Anweisungen eines Blocks sowie im Bedingungsteil von Anweisungen wie `if` und `for` stehen.
  Dadurch ist es möglich, Variablen erst dann zu deklarieren, wenn sie wirklich benötigt werden.
- Variablen können mit *beliebigen* Ausdrücken initialisiert werden.
  Im Gegensatz hierzu dürfen *globale* bzw. *statische* Variablen in C nur mit *konstanten* Ausdrücken (deren Wert vom Compiler zur Übersetzungszeit bestimmt werden kann) initialisiert werden.
- Funktionen und Operatoren können *statisch überladen* werden, d. h. in einer Art und Weise „mehrfach" definiert werden, die vom Compiler zur *Übersetzungszeit* aufgelöst werden kann. Als Spezialfall können Funktionen *optionale Parameter* mit Defaultwerten besitzen.
  Auf diese Weise können verschiedene „Varianten" einer Funktion, die sich in der Anzahl oder in den Typen ihrer Parameter unterscheiden, mit demselben Namen bezeichnet werden. Außerdem ist

es möglich, häufig benötigte Operationen auf benutzerdefinierten Typen durch kompakte und (hoffentlich) natürliche Operatorschreibweisen zu formulieren.

- Funktionen können `inline` deklariert werden.
  Dadurch ist es möglich, die Vorteile echter Funktionen (saubere Schnittstelle, Typsicherheit, lokale Variablen) mit denen von Makros (effizientere Ausführung) zu kombinieren und somit auf die Verwendung von Makros zu verzichten.

- Funktionsparameter können, analog zu `VAR`-Parametern in Modula, *by reference* übergeben werden.
  Auf diese Weise entfällt das lästige und maschinenorientierte Übergeben von Adressen, wie es in C erforderlich ist.

- Datentypen und Funktionen können *Typen als Parameter* besitzen.
  Mit Hilfe solcher *Templates* können Containertypen wie Listen oder Mengen und die zugehörigen Operationen unabhängig von einem bestimmten Elementtyp implementiert und anschließend *typsicher* für beliebige Elementtypen verwendet werden.

- Es gibt Anweisungen zum Erzeugen und Behandeln von *Ausnahmen* (engl. exceptions).
  Damit kann auf unerwartete Fehler während der Programmausführung flexibler reagiert werden, als es mit den Standardanweisungen `return` (zum Verlassen einer Funktion) und `exit` (zum Beenden des ganzen Programms) möglich ist.

- Es gibt einen eigenen Typ `bool` mit den zugehörigen Konstanten `true` und `false`.
  Auf diese Weise können Boolesche Werte (anders als in C) rein „sprachlich" von Integer-Werten unterschieden werden, obwohl die Konstanten `true` und `false` nach wie vor kompatibel zu den Integerwerten `1` (bzw. ungleich null) und `0` sind.

A.2.2 Spracherweiterungen

Darüber hinaus werden mit Hilfe von Präprozessoren und Bibliotheksmodulen (vgl. § A.3) die folgenden *Spracherweiterungen* unterstützt:

- Anstelle von undurchsichtigen C- bzw. C++-`#include`-Anweisungen gibt es ein *Modulkonzept* im Stil von Modula oder Oberon mit expliziten *Import/Export*-Schnittstellen.
  Auf diese Weise kann die Herkunft eines Bezeichners, der in einem bestimmten Modul verwendet wird, unmittelbar aus der Import-Liste dieses Moduls bestimmt werden. Im Gegensatz dazu müssen bei der Verwendung von `#include`-Anweisungen u. U. sämtliche direkt oder indirekt eingebundenen Dateien durchsucht werden.
  Außerdem wird durch ein derartiges Modulkonzept − unabhängig von objektorientierten Konzepten wie Klassen oder Methoden − auf eine einfache und flexible Weise das Prinzip der *Datenabstraktion* bzw. -kapselung unterstützt [Wirth82].

- Anstelle von starren und gefährlichen C-Vektoren (Arrays) auf der einen Seite und oftmals unnötig komplizierten und unhandlichen C++-Containerklassen [Kofler93, Schildt99] auf der anderen Seite, gibt es einen einfachen und doch mächtigen und flexiblen Containertyp *Sequenz*, der alle üblichen Anwendungen von Containern (wie z. B. sortierte oder unsortierte Listen, Stacks, Mengen usw.) gut unterstützt. Insbesondere gibt es ein sehr einfaches *Iterator-Konzept* mit Hilfe einer `forall`-Anweisung.
  Als Spezialfall solcher dynamischer Sequenzen wird ein flexibler *Stringtyp* unterstützt.
  (Vgl. auch § 4.4.2.1, § 4.6.2.3 und § 4.8.1.3 sowie § A.3.2.1 und § A.3.2.2.)

- Anstelle von (varianten) Modula-Records, C-`struct`- bzw. -`union`-Typen oder C++-Klassen gibt es *offene Typen* mit einer erweiterbaren Menge *optionaler Attribute*.
  Auf diese Weise ist es beispielsweise möglich, einen Strukturtyp, der in einem Modul definiert (und exportiert) wird, in einem anderen Modul um zusätzliche Komponenten zu erweitern, ohne daß hierfür das erste Modul geändert oder ein neuer, abgeleiteter Typ eingeführt werden müßte.
  Objekte eines offenen Typs können eine beliebige *Teilmenge* der Attribute des Typs besitzen, wobei

beim Zugriff auf nicht vorhandene Attribute ein definierter Null- bzw. Nil-Wert zurückgeliefert wird.
Durch die Möglichkeit, eindeutige konstante Ausprägungen (engl. instances) eines offenen Typs definieren zu können, stellen diese Typen auch *erweiterbare Aufzählungstypen* dar.
(Vgl. auch § 4.3.3.1, § 4.3.3.2, § 4.4.2.2 und § 4.6 sowie § A.3.2.4.)

- Anstelle einer (i. d. R. mühsamen und fehlerträchtigen) Speicherverwaltung durch den Programmierer gibt es eine *automatische Speicherbereinigung* (engl. garbage collection).[1]

- Durch die Definition einer einzigen Vergleichsprozedur `cmp()` für einen benutzerdefinierten (d. h. in der Regel offenen) Typ werden automatisch die sechs Vergleichsoperatoren `==`, `!=`, `<`, `<=`, `>` und `>=` für diesen Typ definiert. Außerdem ist `cmp()` für alle eingebauten Typen der Sprache einschließlich Sequenzen vordefiniert, so daß neue Vergleichsvorschriften mit geringem Aufwand auf existierende zurückgeführt werden können.
(Vgl. auch § 4.4.2.3 und § 4.8.1.3 sowie § A.3.2.5.)

- Anstelle von virtuellen Funktionen (bzw. äquivalenten Formen von *late binding* in anderen objektorientierten Programmiersprachen) gibt es *partielle Funktionen*, mit deren Hilfe *Laufzeitpolymorphismus* (d. h. die laufzeitabhängige Auswahl einer bestimmten Variante einer Funktion) sowohl flexibler als auch unabhängig von einer Klassenhierarchie realisiert werden kann.
Hierbei können beliebig viele „Varianten“ oder „Teile“ derselben Funktion definiert werden, von denen jede mit einem *Auswahlprädikat* versehen wird, das typischerweise von den formalen Parametern der Funktion abhängt. Zur Laufzeit wird dann immer diejenige Variante der Funktion ausgeführt, deren Prädikat von den aktuellen Aufrufparametern erfüllt wird. Auf diese Weise steht als Auswahlkriterium nicht nur der *Typ* **eines** Objekts zur Verfügung, sondern prinzipiell *beliebige Eigenschaften* **aller** Argumente einer Funktion.
Mathematisch ausgedrückt, stellt jede Variante einer partiellen Funktion einen Teil einer stückweise definierten Funktion dar.
(Vgl. auch § 4.4.2.4, § 4.6.1.2 und § 4.6.4.1 sowie § A.3.2.6.)

- Anstelle der speziellen „Hauptfunktion“ `main()` sowie Bibliotheksmechanismen (wie z. B. `atexit()`) zur Registrierung von „Aufräumfunktionen“ gibt es die Schlüsselwörter `begin` und `end` zur expliziten Kennzeichnung von *Initialisierungs-* bzw. *Terminierungsanweisungen*.
Bei einer normalen Programmausführung werden zunächst (wie in Modula oder Oberon) sämtliche `begin`-Anweisungsblöcke des Programms in einer durch die Import-Beziehungen zwischen den Modulen festgelegten Bottom-up-Reihenfolge ausgeführt. Anschließend werden, in der umgekehrten Reihenfolge, sämtliche `end`-Anweisungen ausgeführt.
(Vgl. auch § 4.6.5 sowie § A.3.2.7.)

A.2.3 Einschränkungen

Die folgenden Konzepte von C und C++ gehören aus der Sicht eines Anwendungsprogrammierers *nicht* zum Sprachumfang von CH und wurden höchstens zur Implementierung von Bibliotheksmodulen verwendet:

- Die gesamte Funktionalität des C- bzw. C++-Präprozessors, also insbesondere die Anweisungen `#define` und `#include`.
Wie bereits erwähnt, werden diese primitiven Konzepte durch `inline`-Funktionen bzw. das Modulkonzept wesentlich besser unterstützt.

- Zeiger und Vektoren.
Durch die konsequente Verwendung von Sequenzen und offenen Typen kann auf diese „berühmt-berüchtigten“ Konzepte vollständig verzichtet werden.

[1] In der aktuellen Sprachversion ist sie allerdings noch nicht implementiert. Wie bereits erwähnt, kann sie prinzipiell auf Bibliotheksebene oder aber durch den Einsatz eines „General-purpose-Garbage-collectors“ wie z. B. [Boehm88] realisiert werden [Jones96]. Da der „Speicherhunger“ der IAA-Implementierung relativ gering ist, stellt das Fehlen der „Müllabfuhr“ bisher kein echtes Problem dar.

- Klassen, Vererbung und virtuelle Funktionen.
  Offene Typen (die man ggf. um das Konzept von *Subtypen* erweitern könnte), gepaart mit partiellen Funktionen, bieten mindestens dieselbe Ausdrucksmächtigkeit wie diese objektorientierten Konzepte, können aber zum Teil flexibler verwendet werden und unterstützen eine stärker mathematisch-funktional orientierte Sichtweise.
- Der widersinnige Typ `void`.
  Funktionen, die keine Parameter besitzen, werden (wie in C++) einfach durch eine leere Parameterliste deklariert, während „Funktionen“, die kein Resultat zurückliefern (d. h. Prozeduren in der Terminologie von Modula), stattdessen mit dem Schlüsselwort `proc` gekennzeichnet werden.
- `static`-Deklarationen auf Modulebene.
  Da durch das Modulkonzept nur solche Variablen und Funktionen eines Moduls nach außen sichtbar sind, die explizit *exportiert* werden, ist das explizite *Verbergen* von Namen mittels `static` überflüssig. (Diese Interpretation des Schlüsselworts `static` hat mit seiner sprachlichen Bedeutung ohnehin nichts zu tun.)
  `static`-Deklarationen innerhalb von Funktionen sind nach wie vor sinnvoll und zulässig.
- Benutzerdefinierte Typumwandlungen.
  Bedauerlicherweise ist dieses Konzept in C++ untrennbar mit dem Klassenkonzept verbunden, so daß es unabhängig davon nicht verwendet werden kann.

A.3 Implementierung der Sprache

A.3.1 Präprozessoren

Ein CH-Programm, d. h. eine Menge von CH-Modulen, die durch Import-Anweisungen miteinander verbunden sind, wird zunächst von einem (eigentlich sprachunabhängigen) *Import-Präprozessor* bearbeitet, dessen Funktionalität grob mit dem Include-Mechanismus des Standard-C-Präprozessors verglichen werden kann. Neben dem rein textuellen Einbinden importierter Module sorgt dieser Präprozessor jedoch auch für eine geeignete *Qualifizierung* aller im Programm verwendeten Bezeichner, um Namenskonflikte zwischen Modulen zu vermeiden.[2]

Anschließend wird das Programm von einem zweiten, CH-*spezifischen Präprozessor* bearbeitet, der die Deklarationen von offenen Typen und Attributen, partiellen Funktionen sowie Initialisierungs- und Terminierungsanweisungen in geeignete C++-Deklarationen umwandelt.

In einem dritten Schritt wird das so vorverarbeitete Programm von einem gewöhnlichen C++-Compiler übersetzt und mit *Bibliotheksmodulen* zur Unterstützung dynamischer Objekte (Sequenzen, Strings, Ausprägungen offener Typen), partieller Funktionen u. ä. zusammengebunden (vgl. § A.3.2).

A.3.2 Bibliotheksmodule

A.3.2.1 Dynamische Sequenzen

Hinter dem generischen Typ `Seq(Elem)` verbirgt sich eine C++-Template-Klasse

```
// Repräsentation einer Sequenz.
template <class Elem> class SeqRep {
    int len;            // Länge (Anzahl Elemente).
    Elem* elems;        // Dynamisches Array von Elementen.
};
```

[2] Dieses Ziel könnte ebenso durch die Verwendung von C++-Namensräumen (engl. namespaces) erreicht werden. Der Import-Präprozessor realisiert das Konzept von Modulen mit expliziten Import/Export-Schnittstellen jedoch *unabhängig* von einer konkreten Programmiersprache und kann daher in gleicher Weise auch für andere Sprachen (wie z. B. C, AWK o. ä.) eingesetzt werden, die das Konzept von Namensräumen nicht unterstützen.

zur Verwaltung dynamischer Arrays mit Elementen vom Typ `Elem`. Diese Art der Repräsentation erlaubt einen direkten und damit sehr schnellen Zugriff auf die einzelnen Elemente einer Sequenz, der jedoch mit relativ ineffizienten Einfüge- und Löschoperationen − insbesondere bei sehr langen Sequenzen − erkauft wird. Daher sollen die flachen Arrays längerfristig durch baumartige Speicherungsstrukturen ersetzt werden, die sowohl Lese- als auch Änderungsoperationen nahezu optimal unterstützen [Boehm95].

Mittels überladener Operatoren unterstützen Sequenzen unter anderem die in Tab. A.1 genannten Operationen. Hinter dem Schlüsselwort `forall` verbirgt sich ein C++-Makro, das den Ausdruck `e << s` unter trickreicher Verwendung überladener Operatoren zur Initialisierung einer gewöhnlichen `for`-Schleife verwendet, die anschließend dafür sorgt, daß die Variable `e` alle Elemente der Sequenz `s` durchläuft. Mit Hilfe weiterer Anweisungen ist es außerdem möglich, das aktuelle Schleifenelement `e` aus der Sequenz `s` zu entfernen, ohne die Iteration zu stören, Iterationen zu unterbrechen und später fortzusetzen usw. Darüber hinaus kann eine `forall`-Schleife einen nachgestellten `else`-Teil besitzen, der genau dann ausgeführt wird, wenn die Schleife komplett durchlaufen, d. h. nicht vorzeitig abgebrochen wurde (vgl. § 4.6.4.2).

Operation	Erläuterung
`*s`	Kardinalität (Anzahl Elemente) der Sequenz `s`
`s[i]`	`i`-tes Element der Sequenz `s`
`s(i, j)`	Teilsequenz vom `i`-ten bis zum `j`-ten Element der Sequenz `s`
`-s`	Sequenz `s` mit invertierter Element-Reihenfolge
`+s`	Kopie der Sequenz `s`
`s += e`	Element `e` am Ende der Sequenz `s` anfügen
`s -= e`	Element `e` aus der Sequenz `s` entfernen
`seq(e1, ..., en)`	Sequenz mit den Elementen `e1`, …, `en`
`forall (e << s) ...`	Iteration über alle Elemente `e` der Sequenz `s`

Tabelle A.1: Sequenz-Operationen (Auswahl)

A.3.2.2 Mengen

Mengen vom Typ `Set(Elem)` werden als *sortierte, duplikatfreie Sequenzen* implementiert, die sich rein syntaktisch nicht von gewöhnlichen Sequenzen unterscheiden (d. h. `Set(Elem)` ist lediglich ein Synonym für `Seq(Elem)`; vgl. auch § 4.6.2.3). Daher unterstützen Sequenzen auch die in Tab. A.2 genannten Mengenoperationen.

Anmerkung: Das Komplement `~s` einer Menge `s` wird implementierungstechnisch durch einen einfachen Indikator (engl. flag) realisiert. Operationen auf einer Komplementmenge `~s` werden nach den üblichen mathematischen Regeln auf Operationen auf der Ausgangsmenge `s` zurückgeführt. Beispielsweise liefert der Elementtest `e << ~s` genau das entgegengesetzte Resultat des Tests `e << s`.

Die Operationen `s += e` (Tab. A.1) und `s |= e` (Tab. A.2) unterscheiden sich dadurch, daß erstere das Element `e` *am Ende* der Sequenz `s` anfügt, während letztere `e` zur *Menge* `s` hinzufügt, d. h. *sortiert* in die Sequenz `s` einfügt. Für die Löschoperation `s -= e` und die Iterationsanweisung `forall (e << s)`, die in beiden Tabellen aufgeführt sind, ist es jedoch unerheblich, ob die Sequenz `s` duplikatfrei und sortiert ist (d. h. eine Menge darstellt) oder nicht.

A.3.2.3 Multimengen

Multimengen vom Typ `Mset(Elem)` werden als Mengen von Paaren repräsentiert, die jeweils aus einem Element vom Typ `Elem` und einer zugehörigen Elementkardinalität vom Typ `int` bestehen. Letz-

Operation	Erläuterung
`s1 | s2` `s1 & s2`	Vereinigung der Mengen `s1` und `s2` Durchschnitt der Mengen `s1` und `s2`
`s1 - s2` `~s`	Differenz der Mengen `s1` und `s2` Komplement der Menge `s`
`s |= e` `s -= e`	Element `e` zur Menge `s` hinzufügen Element `e` aus der Menge `s` entfernen
`s1 << s2` `s1 >> s2`	Ist `s1` Teilmenge von `s2`? Ist `s1` Obermenge von `s2`?
`e << s` `s >> e`	Ist `e` ein Element der Menge `s`? Enthält die Menge `s` das Element `e`?
`set(e1, ..., en)` `forall (e << s) ...`	Menge mit den Elementen `e1`, ..., `en` Iteration über alle Elemente `e` der Menge `s`

Tabelle A.2: Mengenoperationen (Auswahl)

tere bleibt bei den meisten Operationen (wie z. B. `m += e` oder `m -= e` zum Hinzufügen bzw. Entfernen einer Ausprägung des Elements `e`) verborgen, kann aber bei Bedarf gezielt abgefragt werden. Auch die Iteration `forall (e << m)` über alle Elemente `e` der Multimenge `m` unterscheidet sich syntaktisch nicht von einer Iteration über eine Sequenz oder Menge, allerdings wird jedes Element von `m` – unabhängig von seiner Kardinalität – genau einmal durchlaufen. Die Konstante `oo` repräsentiert die spezielle Kardinalität ∞. (Vgl. auch § 4.6.4.2.)

A.3.2.4 Offene Typen

Offene Typen, die u. a. die in Tab. A.3 genannten Operationen unterstützen, werden auf Klassen abgebildet, die von einer generischen Klasse

Operation	Beschreibung
`type T;`	Deklaration des offenen Typs `T`
`attr a: T -> X;` `attr a: T -> X = ...;`	Deklaration des Attributs `a` (vom Typ `X`) des offenen Typs `T` Deklaration eines abgeleiteten (berechneten) Attributs `a`
`inst T { t1, ..., tn };`	Erzeugung eindeutiger Ausprägungen (Aufzählungswerte) `t1, ..., tn` des offenen Typs `T`
`T(a = x, ...)`	Erzeugung eines Objekts vom Typ `T` und Initialisierung des Attributs `a` mit dem Wert `x` (vom Typ `X`)
`o(a = x, ...)`	Zuweisung des Werts `x` (vom Typ `X`) an das Attribut `a` des Objekts `o` (vom Typ `T`)
`o(~a, ...)`	Entfernen des Attributs `a` aus dem Objekt `o` (vom Typ `T`)
`o/a`	Wert (vom Typ `X`) des Attributs `a` des Objekts `o` (vom Typ `T`) bzw. Defaultwert des Typs `X`, falls `o` kein Attribut `a` besitzt
`+o`	Kopie des Objekts `o`

Tabelle A.3: Operationen auf offenen Typen (Auswahl)

```
// Repräsentation eines Objekts (Ausprägung eines offenen Typs).
class ObjRep {
    Sign sign;          // Signatur des Objekts.
    addr data;          // Zeiger auf Datenbereich des Objekts.
};
```

abgeleitet sind. Die Signatur eines Objekts beschreibt hierbei, welche Attribute das Objekt momentan besitzt und an welcher Position (engl. offset) sie im Datenbereich des Objekts gespeichert sind. Wird durch eine Änderungsoperation (wie z. B. `o(a = x)` oder `o(~a)`) ein Attribut `a` dynamisch zu einem Objekt `o` hinzugefügt oder entfernt, so erhält das Objekt eine neue Signatur und sein Datenbereich wird geeignet reorganisiert.

Da in einem laufenden Programm normalerweise viele Objekte mit derselben Signatur existieren, wird eine globale Signaturtabelle verwaltet, die nur bei Bedarf erweitert wird. Auf diese Weise kann der zur Verwaltung von Signaturen benötigte Speicherplatz sehr klein gehalten werden.

Beim Zugriff `o/a` auf ein Attribut `a` des Objekts `o` muß die Signatur des Objekts nach diesem Attribut durchsucht werden, um festzustellen, ob das Attribut existiert, und um ggf. seine Position im Datenbereich zu ermitteln. Da die Attribute einer Signatur nach einer internen Identifikationsnummer sortiert werden, kann dies effizient mittels binärer Suche erfolgen. Dennoch ist ein derartiger Attributzugriff natürlich bei weitem nicht so effizient wie ein Zugriff auf eine gewöhnliche Strukturkomponente, deren relative Position vom Compiler bereits zur Übersetzungszeit bestimmt werden kann.

A.3.2.5 Vergleichsprozeduren und -operatoren

Für atomare Typen berechnet die Vergleichsprozedur `cmp()` die Differenz ihrer beiden Argumente. Ist diese von null verschieden, wird ihr Vorzeichen mittels einer Ausnahme „zurückgeliefert“, andernfalls terminiert die Prozedur normal.

Die sechs Vergleichsoperatoren `==`, `!=`, `<`, `<=`, `>` und `>=` werden als Template-Operatorfunktionen für beliebige Typen definiert. Sie rufen die Prozedur `cmp()` mit ihren beiden Argumenten auf und werten deren „Resultat“ aus: Terminiert `cmp()` normal, werden die beiden Objekte als gleich angesehen; tritt eine Ausnahme auf, so bestimmt das durch sie übermittelte Vorzeichen, welches Objekt das „kleinere“ darstellen soll.

Dieser Mechanismus funktioniert insbesondere bei verschachtelten Aufrufen von `cmp()`, wie sie typischerweise beim Vergleich hierarchischer Objektstrukturen auftreten: Sobald an einer Stelle der Aufrufhierarchie eine von null verschiedene Differenz auftritt (d. h. zwei korrespondierende Teilobjekte verschieden sind), wird eine Ausnahme ausgelöst, die sämtliche Inkarnationen von `cmp()` vorzeitig beendet und das passende Vorzeichen zurückliefert. Terminieren alle Aufrufe von `cmp()` normal, so sind die beiden betrachteten Objektstrukturen gleich.

Als illustrierendes Beispiel betrachte man die generische Definition der Prozedur `cmp()` für Sequenzen vom Typ `Seq(Elem)` in Abb. A.4. Im ersten Schritt werden die Längen (vom Typ `int`) der Sequenzen `s1` und `s2` durch den „rekursiven“ Aufruf `cmp(*s1, *s2)` verglichen. Sind sie verschieden, wird eine entsprechende Ausnahme ausgelöst, die sowohl `cmp(*s1, *s2)` als auch `cmp(s1,`

```
template <class Elem> proc cmp(Seq(Elem) s1, Seq(Elem) s2) {
    // Längen vergleichen.
    cmp(*s1, *s2);

    // Einzelne Elemente vergleichen.
    Elem e1, e2;
    forall2 (e1 << s1, e2 << s2) cmp(e1, e2);
}
```

Abbildung A.4: Vergleich von Sequenzen

s2) (als auch eventuell übergeordnete Aufrufe von cmp()) beendet. Andernfalls terminiert cmp(*s1, *s2) normal, so daß anschließend jeweils korrespondierende Elemente e1 und e2 von s1 bzw. s2 miteinander verglichen werden,[3] wobei abhängig vom konkreten Elementtyp Elem die passende Variante von cmp() verwendet wird (die ihrerseits rekursive Aufrufe von cmp() enthalten könnte usw.). Sobald zwei Elemente (oder Teilobjekte von ihnen) verschieden sind, wird wiederum eine Ausnahme ausgelöst und der gesamte Vergleichsvorgang abgebrochen. Sind alle Elemente paarweise gleich, terminiert cmp(s1, s2) normal, wodurch angezeigt wird, daß s1 und s2 gleich sind.

A.3.2.6 Partielle Funktionen

Jeder Zweig einer partiellen Funktion f() wird auf eine gewöhnliche C++-Funktion (mit einem vom CH-Präprozessor generierten Namen) abgebildet, die vor der Ausführung ihres Rumpfes die Auswahlbedingung des Zweigs überprüft. Ist diese nicht erfüllt, bricht die Funktion mit einer Ausnahme ab.

Auf diese Weise kann die eigentliche Funktion f(), die vom Präprozessor generiert wird, nacheinander jeden ihrer Zweige aufrufen, bis der erste Zweig normal (d. h. ohne die genannte Ausnahme) terminiert. Die Adressen der einzelnen Zweige einer Funktion werden hierfür in einer einfach verketteten Liste gespeichert, die vom Präprozessor mit Hilfe statischer Objektdeklarationen aufgebaut wird.

A.3.2.7 Initialisierungs- und Terminierungsanweisungen

Anweisungsblöcke, die mit einem der Schlüsselworte begin oder end gekennzeichnet sind, werden ebenfalls auf gewöhnliche (parameter- und resultatlose) C++-Funktionen abgebildet, deren Adressen in einer doppelt verketteten Liste gespeichert werden. Die Bibliotheksfunktion main(), die vom C++-Laufzeitsystem nach dem Start des Programms aufgerufen wird, durchläuft diese Liste zunächst „vorwärts“ und ruft hierbei nacheinander die einzelnen Initialisierungsfunktionen (d. h. begin-Blöcke) auf. Anschließend wird die Liste „rückwärts“ durchlaufen und die Terminierungsfunktionen (end-Blöcke) ausgeführt.

A.4 Anmerkungen

Bei der Sprache CH (bzw. ihrer derzeitigen Implementierung) handelt es sich um einen experimentellen Prototyp mit gewissen Einschränkungen bezüglich Robustheit, Fehlertoleranz und Effizienz. Beispielsweise wurden die beiden in § A.3.1 erwähnten Präprozessoren mit relativ geringem Aufwand in der Unix-Skriptsprache AWK [Aho79] geschrieben, was zur Folge hat, daß sie zum einen nicht sehr effizient und zum anderen relativ intolerant in Bezug auf fehlerhafte Eingaben sind.

Bei der Entwicklung der Bibliotheksmodule wurde zwar großer Wert auf Korrektheit, Lesbarkeit und ökonomische Speicherplatzverwendung gelegt, nicht jedoch auf eine maximal effiziente Implementierung. Dies hat zur Folge, daß der Zugriff auf ein Element einer Sequenz oder ein Attribut eines offenen Typs um einen konstanten Faktor in der Größenordnung 1 bis 10 langsamer ist als ein entsprechender Zugriff auf ein Element eines C-Vektors oder eine Komponente eines C-struct’s. Entsprechendes gilt für den Aufruf einer partiellen Funktion, wenn man ihn mit dem Aufruf einer gewöhnlichen C-Funktion vergleicht.

Demgegenüber steht jedoch ein „Effizienzgewinn“ bei der Entwicklung von Programmen, der vermutlich in derselben Größenordnung liegt. Insbesondere wird man durch die Verwendung von Sequenzen und offenen Typen von vielen lästigen, fehleranfälligen und zeitraubenden (Speicher-) Verwaltungsaufgaben befreit und kann sich so ganz auf die eigentlich zu lösenden algorithmischen Probleme konzentrieren.

Da die Sprache CH jedoch nur ein „Abfallprodukt“ dieser Arbeit darstellt und bisher keine wissenschaftlich fundierten Vergleiche mit anderen Programmiersprachen vorliegen, soll die Diskussion über mögliche Vor- und Nachteile ihrer Konzepte hier nicht weiter vertieft werden. Aus der persönlichen

[3] forall2 entspricht forall, erlaubt aber eine gleichzeitige Iteration über zwei Sequenzen, Mengen oder Multimengen.

Sicht des Verfassers wurde die Implementierung von Interaktionsausdrücken durch ihre Verwendung jedoch erheblich vereinfacht.

Anhang B

Verifikation des Zustandsmodells

B.1 Einleitung

Dieser Anhang enthält Hilfssätze und Beweise, die aus Platzgründen aus Kapitel 4 ausgelagert wurden. In § B.2 wird zunächst die Korrektheit des *einfachen Zustandsmodells* (ohne Optimierungsfunktion), wie es in § 4.5 definiert wurde, nachgewiesen, während § B.3 den entsprechenden Beweis für das *optimierte Modell* enthält. § B.4 enthält den Beweis des Satzes über *injektive und fokussierte Ausdrücke* aus § 4.7.2.3.

B.2 Korrektheit des einfachen Zustandsmodells

B.2.1 Vorbereitungen

B.2.1.1 Monotonie relevanter Parameterwerte

Satz

Die Menge $\Omega_w(x, q)$ der relevanten Parameterwerte eines zusammengesetzten Ausdrucks

$$x \equiv \bigcirc\, y \qquad \text{oder} \qquad x \equiv y \bigcirc z \qquad \text{oder} \qquad x \equiv \underset{p}{\bigcirc}\, y$$

bzgl. eines Quantorparameters $q \in \Pi$ ist *monoton* bzgl. w und x, d. h. für jedes Teilwort w' von w und jeden Teilausdruck x' von x gilt:

$$\Omega_{w'}(x', q) \subseteq \Omega_w(x, q),$$

genauer:

$$\begin{array}{llll} \Omega_{w'}(y, q) \subseteq \Omega_w(x, q) & & & \text{für } x \equiv \bigcirc\, y, \\ \Omega_{w'}(y, q) \subseteq \Omega_w(x, q) & \text{und} & \Omega_{w'}(z, q) \subseteq \Omega_w(x, q) & \text{für } x \equiv y \bigcirc z, \\ \Omega_{w'}(y, q) \subseteq \Omega_w(x, q) & \text{und} & \Omega_{w'}\left(y_p^{\omega}, q\right) \subseteq \Omega_w(x, q) \text{ für alle } \omega \in \Omega & \text{für } x \equiv \underset{p}{\bigcirc}\, y. \end{array}$$

Beweis

1. Für einen Wert $\pi \in \Omega$ und einen Ausdruck x' mit $\alpha(x') \subseteq \alpha(x)$ gilt die folgende Kette von Implikationen:[1]

   $\pi \in \Omega_{w'}(x', q)$

   ⇓ Definition der relevanten Parameterwerte eines Wortes.

   $\Rightarrow \quad \exists\, a \in w' \colon \pi \in \Omega_a(x', q)$

   ⇓ Definition der relevanten Parameterwerte einer Aktion.

   $\Rightarrow \quad \exists\, a \in w' \colon a \in \left(\alpha(x') \setminus \{ a \}\right)_q^{\pi} \qquad (1)$

   ⇓ $w' \subseteq w \quad \text{und} \quad \alpha(x') \subseteq \alpha(x)$

[1] Die Notationen $a \in w'$ und $w' \subseteq w$ für Aktionen $a \in \Sigma$ und Worte $w', w \in \Sigma^*$ werden im folgenden mit ihrer intuitiven Bedeutung verwendet (w' enthält die Aktion a bzw. w enthält alle Aktionen von w'), obwohl Worte strenggenommen keine Mengen darstellen.

$\Rightarrow \quad \exists\, a \in w : a \in \left(\alpha(x) \setminus \{\, a \,\}\right)_q^\pi \qquad (2)$

⇓ Definition der relevanten Parameterwerte einer Aktion.

$\Rightarrow \quad \exists\, a \in w : \pi \in \Omega_a(x, q)$

⇓ Definition der relevanten Parameterwerte eines Wortes.

$\Rightarrow \quad \pi \in \Omega_w(x, q).$

Damit ist die Behauptung für die Teilausdrücke y und ggf. z eines elementaren Ausdrucks x sowie für die konkretisierten Teilausdrücke y_p^ω eines Quantorausdrucks x gezeigt, da deren Alphabet jeweils im Alphabet von x enthalten ist.

2. Für den abstrakten Zweig y eines Quantorausdrucks $x \equiv \underset{p}{\bigcirc}\, y$ gilt:

$$a \in \left(\alpha(y) \setminus \{\, a \,\}\right)_q^\pi$$

$\Rightarrow \quad \exists\, \tilde{a} \in \alpha(y) \setminus \{\, a \,\} : \tilde{a}_q^\pi = a \in \Sigma$

⇓ Wegen $\tilde{a}_q^\pi \in \Sigma$ muß $\tilde{a}$ unabhängig vom Parameter p sein,
⇓ d. h. es gilt: $\tilde{a} = \tilde{a}_p^\omega$ für alle $\omega \in \Omega$.

$\Rightarrow \quad \exists\, \tilde{a} = \tilde{a}_p^\omega \in \alpha\left(y_p^\omega\right) \setminus \{\, a \,\} : \tilde{a}_q^\pi = a$

⇓ $\alpha\left(y_p^\omega\right) \subseteq \alpha(x)$

$\Rightarrow \quad \exists\, \tilde{a} \in \alpha(x) \setminus \{\, a \,\} : \tilde{a}_q^\pi = a$

$\Rightarrow \quad a \in \left(\alpha(x) \setminus \{\, a \,\}\right)_q^\pi.$

Fügt man diese Zwischenschritte oben zwischen Zeile (1) und (2) ein, so erhält man die verbleibende Behauptung.

B.2.1.2 Invarianz relevanter Parameterwerte

Satz

Gegeben sei ein Quantorausdruck $x \equiv \underset{p}{\bigcirc}\, y$, ein Wort $w \in \Sigma^*$, ein Quantorparameter $q \in \Pi$ sowie ein bzgl. x und q irrelevanter Parameterwert $\pi \notin \Omega_w(x, q)$.

Dann gilt:

$$\Omega_w\left(y_q^\pi,\, p\right) = \Omega_w(y,\, p),$$

d. h. die relevanten Parameterwerte des Quantorrumpfs y ändern sich nicht, wenn man y zu y_q^π konkretisiert.

Beweis

Für eine Aktion $a \in w$ und einen Wert $\omega \in \Omega$ gilt die folgende Kette von Äquivalenzen:

$$\omega \in \Omega_a\left(y_q^\pi,\, p\right)$$

$\Downarrow$ Definition der relevanten Parameterwerte einer Aktion.

$$\Leftrightarrow \quad a \in \left(\alpha\left(y_q^\pi\right) \setminus \{\, a \,\}\right)_p^\omega$$

$$\Leftrightarrow \quad \exists\, \tilde{a} \in \alpha\left(y_q^\pi\right) \setminus \{\, a \,\}\colon \tilde{a}_p^\omega = a$$

$$\Leftrightarrow \quad \exists\, \tilde{a} \in \alpha\left(y_q^\pi\right)\colon \tilde{a} \neq a = \tilde{a}_p^\omega$$

$$\Leftrightarrow \quad \exists\, \tilde{\tilde{a}} \in \alpha(y),\ \tilde{a} = \tilde{\tilde{a}}_q^\pi\colon \tilde{a} \neq a = \tilde{a}_p^\omega$$

$$\Leftrightarrow \quad \exists\, \tilde{\tilde{a}} \in \alpha(y)\colon \tilde{\tilde{a}}_q^\pi \neq a = \tilde{\tilde{a}}_{p,\,q}^{\omega,\,\pi}$$

$\Downarrow$ Es gilt: $\tilde{\tilde{a}} \in \alpha(y) \quad \Rightarrow \quad \tilde{\tilde{a}}_p^\omega \in \alpha\left(y_p^\omega\right) \subseteq \alpha(x) \quad \Rightarrow \quad a = \tilde{\tilde{a}}_{p,\,q}^{\omega,\,\pi} \in (\alpha(x))_q^\pi$.
$\Downarrow$ Annahme: $\tilde{\tilde{a}}_p^\omega \neq a \quad \Rightarrow \quad \tilde{\tilde{a}}_p^\omega \in \alpha(x) \setminus \{\, a \,\} \quad \Rightarrow$
$\Downarrow$ $a = \tilde{\tilde{a}}_{p,\,q}^{\omega,\,\pi} \in \left(\alpha(x) \setminus \{\, a \,\}\right)_q^\pi \quad \Rightarrow \quad \pi \in \Omega_a(x,\, q)$,
$\Downarrow$ im Widerspruch zur Voraussetzung $\pi \notin \Omega_w(x,\, q)$.
$\Downarrow$ Daher gilt: $\tilde{\tilde{a}}_p^\omega = a \in \Sigma$, d. h. $\tilde{\tilde{a}}$ ist unabhängig vom Parameter q, woraus $\tilde{\tilde{a}} = \tilde{\tilde{a}}_q^\pi$ folgt.

$$\Leftrightarrow \quad \exists\, \tilde{\tilde{a}} \in \alpha(y)\colon \tilde{\tilde{a}} \neq a = \tilde{\tilde{a}}_p^\omega$$

$$\Leftrightarrow \quad \exists\, \tilde{\tilde{a}} \in \alpha(y) \setminus \{\, a \,\}\colon \tilde{\tilde{a}}_p^\omega = a$$

$$\Leftrightarrow \quad a \in \left(\alpha(y) \setminus \{\, a \,\}\right)_p^\omega$$

$\Downarrow$ Definition der relevanten Parameterwerte einer Aktion.

$$\Leftrightarrow \quad \omega \in \Omega_a(y,\, p),$$

d. h. $\Omega_a\left(y_q^\pi,\, p\right) = \Omega_a(y,\, p)$, woraus unmittelbar die Behauptung folgt.

B.2.2 Korrektheitstheorem

B.2.2.1 Satz

Das in § 4.5 definierte *einfache Zustandsmodell* für Interaktionsausdrücke ist *korrekt* und erfüllt das in § 4.5.6.3 genannte *Substitutionsprinzip*, d. h. für einen Ausdruck x und ein Wort $w \in \Sigma^*$ gilt:

1. $w \in \Psi(x) \Leftrightarrow \psi_w(x) = \top$ und $w \in \Phi(x) \Leftrightarrow \varphi_w(x) = \top$ (Korrektheit).
2. $\sigma_w\left(x_q^\pi\right) = \left(\sigma_w(x)\right)_q^\pi$ für $\pi \notin \Omega_w(x,\, q)$ (Substitutionsprinzip).

B.2.2.2 Beweisstruktur

Der Beweis dieses Satzes erfolgt mittels *Induktion* nach der Struktur des Ausdrucks x (vgl. § 3.4.3), d. h. für zusammengesetzte Ausdrücke x wird die Richtigkeit der Behauptungen für die Teilausdrücke von x bereits vorausgesetzt („globale Induktionsvoraussetzung“).

Ähnlich wie die Definition des Zustandsmodells in § 4.5, ist auch der Beweis nach der Kategorie des Ausdrucks x partitioniert. Der Beweis für atomare Ausdrücke x in § B.2.3.1 stellt hierbei den *Induktionsanfang* dar, während die Beweise für die übrigen Ausdruckskategorien in den Abschnitten B.2.3.2 bis B.2.3.8 und B.2.4.1 bis B.2.4.4 zusammen den *Induktionsschritt* darstellen.

Um die Behauptungen zu zeigen, wird in jedem dieser Abschnitte zunächst eine Aussage über die *Struktur der Folgezustände* $\sigma_w(x)$ des Ausdrucks x formuliert und − sofern ihre Richtigkeit nicht unmittelbar offensichtlich ist − mittels vollständiger Induktion nach der Länge des Wortes w bewiesen.

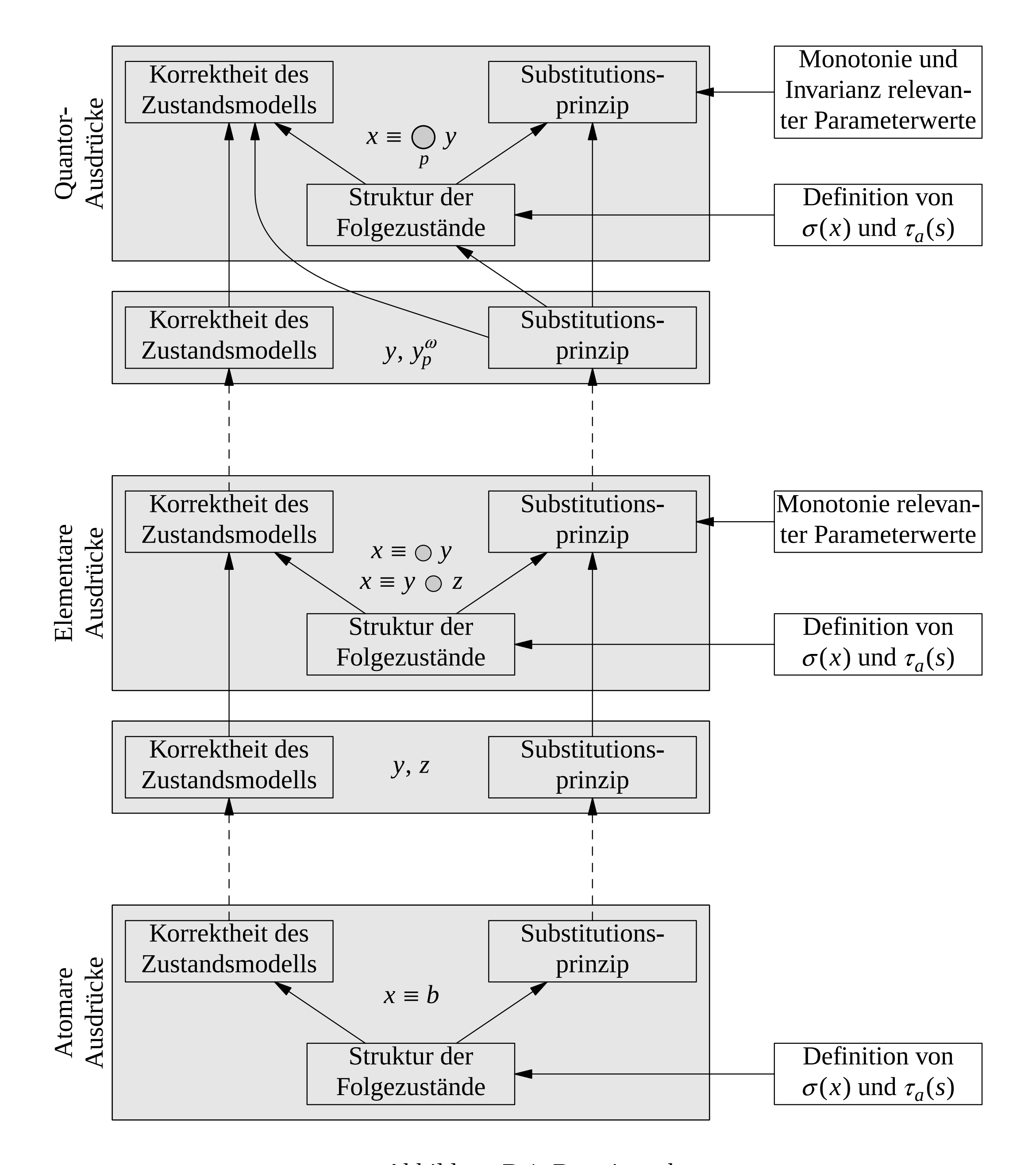


Abbildung B.1: Beweisstruktur

Diese „lokalen" Induktionsbeweise, bei denen der Fall $w = \langle\rangle$ den Induktionsanfang und der Fall $w = \tilde{w}\langle a\rangle$ (mit $\tilde{w} \in \Sigma^*$ und $a \in \Sigma$) den Induktionsschritt darstellt, haben jedoch nichts mit dem zuvor erwähnten „globalen Induktionsbeweis" nach der Struktur des Ausdrucks x zu tun.

Abbildung B.1 veranschaulicht die Zusammenhänge zwischen den einzelnen Schritten des globalen Beweises. Für atomare Ausdrücke $x \equiv b$ (unten) läßt sich die Korrektheit des Zustandsmodells und die Gültigkeit des Substitutionsprinzips direkt aus der Aussage über die Struktur der Folgezustände herleiten, während man für elementare Ausdrücke $x \equiv \bigcirc\, y$ und $x \equiv y \bigcirc z$ (Mitte) zusätzlich die Korrektheit des Zustandsmodells bzw. die Gültigkeit des Substitutionsprinzips für die Teilausdrücke y und ggf. z benötigt. Für Quantorausdrücke $x \equiv \underset{p}{\bigcirc}\, y$ (oben) wird das Substitutionsprinzip für den Quantorrumpf y außerdem benötigt, um die Aussage über die Struktur der Folgezustände und die Korrektheit des Zustandsmodells nachzuweisen. (Dies ist der eigentliche Grund für die Notwendigkeit des Substitutionsprinzips.)

Die Aussagen über die Struktur der Folgezustände basieren einerseits auf der Definition des initialen Zustands $\sigma(x)$ und andererseits auf der Definition des Zustandsübergangs $\tau_a(s)$. Bei der Anwendung des Substitutionsprinzips auf Teilausdrücke wird implizit die Monotonie relevanter Parameterwerte ausgenutzt (§ B.2.1.1), die gewährleistet, daß aus der Voraussetzung $\pi \notin \Omega_w(x, q)$ folgt: $\pi \notin \Omega_{w'}\big(x', q\big)$ für jedes Teilwort w' von w und jeden Teilausdruck x' von x. In den Beweis des Substitutionsprinzips für Quantorausdrücke fließt außerdem die Invarianz relevanter Parameterwerte ein (§ B.2.1.2).

B.2.3 Beweis für elementare Ausdrücke

B.2.3.1 Atomare Ausdrücke

Gegeben sei ein atomarer Ausdruck $x \equiv b$ mit einer abstrakten Aktion $b \in \Gamma$.

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt offensichtlich:

$$\sigma_w(x) = \begin{cases} b & \text{für } w = \langle\rangle, \\ \top & \text{für } w = \langle b \rangle, \\ \bot & \text{sonst,} \end{cases}$$

und somit:

$$\psi_w(x) = \begin{cases} \top & \text{für } w \in \{\, \langle\rangle, \langle b \rangle \,\} = \Psi(x), \\ \bot & \text{sonst,} \end{cases} \qquad \text{und} \qquad \varphi_w(x) = \begin{cases} \top & \text{für } w \in \{\, \langle b \rangle \,\} = \Phi(x), \\ \bot & \text{sonst.} \end{cases}$$

Korrektheit des Zustandsmodells

Die Gültigkeit der Korrektheitskriterien

$$w \in \Psi(x) \;\Leftrightarrow\; \psi_w(x) = \top \qquad \text{und} \qquad w \in \Phi(x) \;\Leftrightarrow\; \varphi_w(x) = \top$$

folgt unmittelbar aus der Struktur der Folgezustände.

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w können die folgenden Fälle unterschieden werden:

1. Für $w = \langle\rangle$ gilt offensichtlich:

   $$\big(\sigma_w(x)\big)_q^\pi = b_q^\pi = \sigma_w\big(x_q^\pi\big).$$

2. Für $w = \langle b \rangle$ muß (wegen $w \in \Sigma^*$) $b \in \Sigma$ und daher (da b folglich unabhängig vom Parameter q ist) $b_q^\pi = b$ und somit $w = \langle b_q^\pi \rangle$ gelten. Daraus folgt aber:

   $$\big(\sigma_w(x)\big)_q^\pi = \top_q^\pi = \top = \sigma_w\big(x_q^\pi\big).$$

3. Für $w = \langle b_q^\pi \rangle$ folgt durch Widerspruch ebenfalls, daß $b = b_q^\pi$ und somit $w = \langle b \rangle$ gilt. Aus der Annahme $b \neq b_q^\pi =: \tilde{b}$ würde nämlich folgen:

   $$\Omega_w(x, q) = \Omega_{\tilde{b}}(x, q) = \Big\{\, \omega \in \Omega \;\Big|\; \tilde{b} \in \big(\alpha(x) \setminus \{\, \tilde{b} \,\}\big)_q^\omega \,\Big\} = \Big\{\, \omega \in \Omega \;\Big|\; \tilde{b} \in \big(\{\, b \,\} \setminus \{\, \tilde{b} \,\}\big)_q^\omega \,\Big\} =$$

   $$\Big\{\, \omega \in \Omega \;\Big|\; \tilde{b} \in \{\, b \,\}_q^\omega \,\Big\} = \Big\{\, \omega \in \Omega \;\Big|\; b_q^\pi \in \big\{\, b_q^\omega \,\big\} \,\Big\} \ni \pi,$$

im Gegensatz zur Voraussetzung $\pi \notin \Omega_w(x, q)$. Daher gilt wie im Fall 2:

$$\left(\sigma_w(x)\right)_q^\pi = \top_q^\pi = \top = \sigma_w\left(x_q^\pi\right).$$

4. Für alle anderen Worte $w \in \Sigma^*$ gilt offensichtlich:

$$\left(\sigma_w(x)\right)_q^\pi = \bot_q^\pi = \bot = \sigma_w\left(x_q^\pi\right).$$

Somit gilt für alle $w \in \Sigma^*$ die Beziehung:

$$\left(\sigma_w(x)\right)_q^\pi = \sigma_w\left(x_q^\pi\right).$$

B.2.3.2 Disjunktion

Gegeben sei eine Disjunktion $x \equiv y \bigcirc z$ mit beliebigen Teilausdrücken y und z, für die das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt offensichtlich (Beweis unmittelbar durch vollständige Induktion nach der Länge des Wortes w):

$$\sigma_w(x) = [\bigcirc, \sigma_w(y), \sigma_w(z)],$$

und somit:

$$\psi_w(x) = \psi_w(y) \vee \psi_w(z) \quad \text{und} \quad \varphi_w(x) = \varphi_w(y) \vee \varphi_w(z).$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) = \Psi(y) \cup \Psi(z)$$

$$\Leftrightarrow \quad w \in \Psi(y) \quad \text{oder} \quad w \in \Psi(z)$$

⇓ Korrektheit des Zustandsmodells für y und z.

$$\Leftrightarrow \quad \psi_w(y) = \top \quad \text{oder} \quad \psi_w(z) = \top$$

⇓ Struktur der Folgezustände von x.

$$\Leftrightarrow \quad \psi_w(x) = \psi_w(y) \vee \psi_w(z) = \top,$$

und analog:

$$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top.$$

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$$\left(\sigma_w(x)\right)_q^\pi = [\bigcirc, \sigma_w(y), \sigma_w(z)]_q^\pi$$

⇓ Definition konkretisierter Zustände (§ 4.5.6.1).

$$= \left[\bigcirc, \left(\sigma_w(y)\right)_q^\pi, \left(\sigma_w(z)\right)_q^\pi\right]$$

⇓ Substitutionsprinzip für y und z (plus Monotonie relevanter Parameterwerte).

$= \left[\bigcirc, \sigma_w\left(y_q^\pi\right), \sigma_w\left(z_q^\pi\right)\right]$

⇓ Struktur der Folgezustände des Ausdrucks $x_q^\pi \equiv y_q^\pi \bigcirc z_q^\pi$.

$= \sigma_w\left(x_q^\pi\right)$.

B.2.3.3 Option

Gegeben sei eine Option $x \equiv \circlearrowright y$ mit einem beliebigen Teilausdruck y, für den das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Struktur der Folgezustände

Da der Ausdruck x als spezielle Disjunktion $x = y \bigcirc \varepsilon$ mit $\sigma(\varepsilon) = \top$ implementiert wird, gilt für seine Folgezustände offensichtlich:

$$\sigma_w(x) = [\bigcirc, \sigma_w(y), \sigma_w(\varepsilon)] \quad \text{mit} \quad \sigma_w(\varepsilon) = \begin{cases} \top & \text{für } w = \langle\rangle, \\ \bot & \text{sonst,} \end{cases}$$

und somit:

$$\psi_w(x) = \psi_w(y) \vee \left(w = \langle\rangle\right) \quad \text{und} \quad \varphi_w(x) = \varphi_w(y) \vee \left(w = \langle\rangle\right).$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$w \in \Psi(x) = \Psi(y) = \Psi(y) \cup \{\langle\rangle\}$

$\Leftrightarrow \quad w \in \Psi(y) \quad \text{oder} \quad w = \langle\rangle$

⇓ Korrektheit des Zustandsmodells für y.

$\Leftrightarrow \quad \psi_w(y) = \top \quad \text{oder} \quad w = \langle\rangle$

⇓ Struktur der Folgezustände von x.

$\Leftrightarrow \quad \psi_w(x) = \psi_w(y) \vee \left(w = \langle\rangle\right) = \top$,

und analog:

$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top$.

Gültigkeit des Substitutionsprinzips

Da das Substitutionsprinzip für den leeren Ausdruck ε offensichtlich erfüllt ist, folgt seine Gültigkeit für den Ausdruck x unmittelbar aus § B.2.3.2.

B.2.3.4 Konjunktion

Da das Zustandsmodell einer Konjunktion $x \equiv y \bullet z$ vollkommen analog bzw. dual zu dem einer Disjunktion definiert ist, ergibt sich auch die Struktur der Folgezustände von x sowie die Korrektheit des Zustandsmodells und die Gültigkeit des Substitutionsprinzips analog.

B.2.3.5 Synchronisation

Gegeben sei eine Synchronisation $x \equiv y \odot z$ mit beliebigen Teilausdrücken y und z, für die das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Definition

Die *Division* w/A eines Wortes $w \in \Sigma^*$ durch eine Aktionsmenge $A \subseteq \Sigma$ sei definiert durch:

$$w/A = \begin{cases} \langle\rangle & \text{für } w = \langle\rangle, \\ \tilde{w}/A & \text{für } w = \tilde{w}\,\langle a\rangle \text{ und } a \in A, \\ (\tilde{w}/A)\,\langle a\rangle & \text{für } w = \tilde{w}\,\langle a\rangle \text{ und } a \notin A. \end{cases}$$

Das Wort w/A ergibt sich also, indem man aus dem Wort w alle Aktionen $a \in A$ *entfernt*.

Lemma

Für ein Wort $w \in u \otimes v$ mit $u \in U^*$, $v \in V^*$ und disjunkten Mengen $U, V \subseteq \Sigma$ gilt:

$$w/V = u,$$

d. h. in diesem Fall stellt die Division w/V quasi die inverse Operation der Verschränkung $u \otimes v$ dar.

Beweis

Wenn man aus einem Wort $w = u_1\,v_1 \ldots u_n v_n$ mit $u_1 \ldots u_n = u$ und $v_1 \ldots v_n = v$ (vgl. § 3.2.2.3) alle Aktionen $a \in V$ entfernt, so entfernt man dadurch (wegen $U \cap V = \emptyset$) genau die Aktionen von v, d. h. die Teilworte $v_1, \ldots, v_n$. Somit verbleiben genau die Teilworte $u_1, \ldots, u_n$ (in dieser Reihenfolge), d. h. das Wort u.

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt:

$$\sigma_w(x) = \big[\,\mathbf{o},\, y,\, z,\, \sigma_{w_y}(y),\, \sigma_{w_z}(z)\big] \quad \text{mit} \quad w_y = w/\kappa_x(y) \quad \text{und} \quad w_z = w/\kappa_x(z),$$

und somit:

$$\psi_w(x) = \psi_{w_y}(y) \wedge \psi_{w_z}(z) \quad \text{und} \quad \varphi_w(x) = \varphi_{w_y}(y) \wedge \varphi_{w_z}(z).$$

Beweis

Vollständige Induktion nach der Länge des Wortes w:

1. Für das leere Wort $w = \langle\rangle$ gilt wegen $w_y = w_z = \langle\rangle$:

   $$\sigma_{w_y}(y) = \sigma(y) \quad \text{und} \quad \sigma_{w_z}(z) = \sigma(z).$$

   Daher stimmt die Behauptung für $\sigma_w(x)$ genau mit der Definition des initialen Zustands $\sigma(x)$ überein.

2. Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

   $$\sigma_w(x) = \tau_a\big(\sigma_{\tilde{w}}(x)\big)$$

   ⇓ Induktionsvoraussetzung für $\sigma_{\tilde{w}}(x)$.

   $$= \tau_a\big([\,\mathbf{o},\, y,\, z,\, l,\, r]\big) \quad \text{mit} \quad l = \sigma_{\tilde{w}_y}(y), \quad \tilde{w}_y = \tilde{w}/\kappa_x(y)$$
   $$\text{und} \quad r = \sigma_{\tilde{w}_z}(z), \quad \tilde{w}_z = \tilde{w}/\kappa_x(z)$$

   ⇓ Definition des Zustandsübergangs τ_a.

$= [\circ, y, z, l', r']$ mit $l' = \begin{cases} l & \text{für } a \in \alpha(z) \setminus \alpha(y) = \kappa_x(y), \\ \tau_a(l) & \text{sonst,} \end{cases}$

und $r' = \begin{cases} r & \text{für } a \in \alpha(y) \setminus \alpha(z) = \kappa_x(z), \\ \tau_a(r) & \text{sonst} \end{cases}$

⇓ Es gilt: $w_y = (\tilde{w}\langle a\rangle)/\kappa_x(y) = \tilde{w}_y\,(\langle a\rangle/\kappa_x(y)) = \begin{cases} \tilde{w}_y & \text{für } a \in \kappa_x(y), \\ \tilde{w}_y\langle a\rangle & \text{sonst.} \end{cases}$

⇓ Daraus folgt: $l' = \begin{cases} l = \sigma_{\tilde{w}_y}(y) = \sigma_{w_y}(y) & \text{für } a \in \kappa_x(y), \\ \tau_a(l) = \tau_a\big(\sigma_{\tilde{w}_y}(y)\big) = \sigma_{\tilde{w}_y\langle a\rangle}(y) = \sigma_{w_y}(y) & \text{sonst,} \end{cases}$

⇓ d. h. es gilt generell $l' = \sigma_{w_y}(y)$, und analog $r' = \sigma_{w_z}(z)$.

$= \big[\circ, y, z, \sigma_{w_y}(y), \sigma_{w_z}(z)\big].$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$w \in \Psi(x) = \Psi(y) \otimes \kappa_x(y)^* \cap \Psi(z) \otimes \kappa_x(z)^*$

$\Leftrightarrow \quad w \in \Psi(y) \otimes \kappa_x(y)^* \quad \text{und} \quad w \in \Psi(z) \otimes \kappa_x(z)^*$

$\Leftrightarrow \quad \exists\, u_y \in \Psi(y), v_y \in \kappa_x(y)^* : w \in u_y \otimes v_y \quad \text{und} \quad \exists\, u_z \in \Psi(z), v_z \in \kappa_x(z)^* : w \in u_z \otimes v_z$

⇓ Aus $w \in u_y \otimes v_y$ mit $u_y \in \Psi(y) \subseteq \alpha(y)^* =: U^*$ und $v_y \in \kappa_x(y)^* =: V^*$ folgt wegen
⇓ $U \cap V = \emptyset$ aufgrund des Lemmas:
⇓ $u_y = w/V = w/\kappa_x(y) = w_y$, und analog $u_z = w_z$.

$\Leftrightarrow \quad w_y = w/\kappa_x(y) \in \Psi(y) \quad \text{und} \quad w_z = w/\kappa_x(z) \in \Psi(z)$

⇓ Korrektheit des Zustandsmodells für y und z.

$\Leftrightarrow \quad \psi_{w_y}(y) = \top \quad \text{und} \quad \psi_{w_z}(z) = \top$

⇓ Struktur der Folgezustände von x.

$\Leftrightarrow \quad \psi_w(x) = \psi_{w_y}(y) \wedge \psi_{w_z}(z) = \top,$

und analog:

$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top.$

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$\big(\sigma_w(x)\big)_q^\pi = \big[\circ, y, z, \sigma_{w_y}(y), \sigma_{w_z}(z)\big]_q^\pi$

⇓ Definition konkretisierter Zustände (§ 4.5.6.1).

$= \Big[\circ, y_q^\pi, z_q^\pi, \big(\sigma_{w_y}(y)\big)_q^\pi, \big(\sigma_{w_z}(z)\big)_q^\pi\Big]$

⇓ Substitutionsprinzip für y und z (plus Monotonie relevanter Parameterwerte).
⇓ (Beachte aber, daß w_y und w_z zunächst unverändert bleiben.)

$= \left[\circ, y_q^\pi, z_q^\pi, \sigma_{w_y}\left(y_q^\pi\right), \sigma_{w_z}\left(z_q^\pi\right)\right]$

⇓ Für eine Aktion $a \in \Sigma$ des Wortes w gilt einerseits:[2] $a \in \alpha(y) \;\Rightarrow\; a = a_q^\pi \in \alpha\left(y_q^\pi\right)$,
⇓ und andererseits:[3] $a \in \alpha\left(y_q^\pi\right) \;\Rightarrow\; \exists\, \tilde{a} \in \alpha(y){:}\ \tilde{a}_q^\pi = a \;\Rightarrow\; a = \tilde{a} \in \alpha(y)$,
⇓ das heißt: $a \in \alpha(y) \;\Leftrightarrow\; a \in \alpha\left(y_q^\pi\right)$, und analog: $a \in \alpha(z) \;\Leftrightarrow\; a \in \alpha\left(z_q^\pi\right)$.
⇓ Wegen $\alpha(x) = \alpha(y) \cup \alpha(z)$ und $\alpha(\tilde{x}) = \alpha(\tilde{y}) \cup \alpha(\tilde{z})$ für $\tilde{x} \equiv x_q^\pi \equiv y_q^\pi \circ z_q^\pi \equiv \tilde{y} \circ \tilde{z}$
⇓ gilt daher auch: $a \in \alpha(x) \;\Leftrightarrow\; a \in \alpha(\tilde{x})$,
⇓ und somit: $a \in \kappa_x(y) \;\Leftrightarrow\; a \in \kappa_{\tilde{x}}(\tilde{y})$ und $a \in \kappa_x(z) \;\Leftrightarrow\; a \in \kappa_{\tilde{x}}(\tilde{z})$.
⇓ Daraus folgt schließlich: $w_y = w/\kappa_x(y) = w/\kappa_{\tilde{x}}(\tilde{y}) = w_{\tilde{y}}$
⇓ und: $w_z = w/\kappa_x(z) = w/\kappa_{\tilde{x}}(\tilde{z}) = w_{\tilde{z}}$.

$= \left[\circ, \tilde{y}, \tilde{z}, \sigma_{w_{\tilde{y}}}(\tilde{y}), \sigma_{w_{\tilde{z}}}(\tilde{z})\right]$

⇓ Struktur der Folgezustände des Ausdrucks $\tilde{x} \equiv \tilde{y} \circ \tilde{z}$.

$= \sigma_w(\tilde{x}) = \sigma_w\left(x_q^\pi\right)$.

B.2.3.6 Sequentielle Komposition

Gegeben sei eine sequentielle Komposition $x \equiv y - z$ mit beliebigen Teilausdrücken y und z, für die das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Definition

Die Menge $\Delta_w^-(y)$ der *sequentiellen Reste* des Wortes w bzgl. des Teilausdrucks y sei definiert als:

$$\Delta_w^-(y) = \{\, v \in \Sigma^* \mid \exists\, u \in \Sigma^*{:}\ w = u\,v,\ \varphi_u(y) \,\}.$$

$\Delta_w^-(y)$ enthält also diejenigen Suffixe v von w, deren „komplementäres Präfix“ u ein vollständiges Wort von y darstellt.

Lemma

Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

$$\Delta_w^-(y) = \{\, \tilde{v}\,\langle a\rangle \mid \tilde{v} \in \Delta_{\tilde{w}}^-(y) \,\} \cup \{\, \langle\rangle \mid \varphi_w(y) \,\}.$$

Beweis

$\Delta_w^-(y) = \{\, v \in \Sigma^* \mid \exists\, u \in \Sigma^*{:}\ w = u\,v,\ \varphi_u(y) \,\}$

⇓ Aus $u\,v = w$ und $w = \tilde{w}\,\langle a\rangle$ folgt entweder $u = w$ und $v = \langle\rangle$
⇓ oder aber $v = \tilde{v}\,\langle a\rangle$ für ein geeignetes $\tilde{v} \in \Sigma^*$.

$= \{\, \tilde{v}\,\langle a\rangle \mid \exists\, u \in \Sigma^*{:}\ w = u\,\tilde{v}\,\langle a\rangle,\ \varphi_u(y) \,\} \cup \{\, \langle\rangle \mid \varphi_w(y) \,\}$

⇓ $w = u\,\tilde{v}\,\langle a\rangle$ ist äquivalent zu $\tilde{w} = u\,\tilde{v}$.

$= \{\, \tilde{v}\,\langle a\rangle \mid \exists\, u \in \Sigma^*{:}\ \tilde{w} = u\,\tilde{v},\ \varphi_u(y) \,\} \cup \{\, \langle\rangle \mid \varphi_w(y) \,\}$

⇓ Definition von $\Delta_{\tilde{w}}^-(y)$.

$= \{\, \tilde{v}\,\langle a\rangle \mid \tilde{v} \in \Delta_{\tilde{w}}^-(y) \,\} \cup \{\, \langle\rangle \mid \varphi_w(y) \,\}$.

[2] Wegen $a \in \Sigma$ kann a nicht vom Parameter q abhängen, d. h. es gilt $a = a_q^\pi$.
[3] Die Gleichheit $a = \tilde{a}$ ergibt sich wie folgt durch Widerspruch: Aus der Annahme $a \neq \tilde{a}$ folgt zunächst $\tilde{a} \in \alpha(y) \setminus \{\, a \,\} \subseteq \alpha(x) \setminus \{\, a \,\}$, und hieraus $a = \tilde{a}_q^\pi \in \left(\alpha(x) \setminus \{\, a \,\}\right)_q^\pi$, was im Widerspruch zur Voraussetzung $\pi \notin \Omega_a(x, q) \subseteq \Omega_w(x, q)$ steht.

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt:

$$\sigma_w(x) = [-, z, \sigma_w(y), R_w] \quad \text{mit} \quad R_w = \{ \sigma_v(z) \mid v \in \Delta_w^-(y) \},$$

und somit:

$$\psi_w(x) = \psi_w(y) \vee \bigvee_{v \in \Delta_w^-(y)} \psi_v(z) \quad \text{und} \quad \varphi_w(x) = \bigvee_{v \in \Delta_w^-(y)} \varphi_v(z).$$

Beweis

1. Für das leere Wort $w = \langle\rangle$ gilt $\sigma_w(y) = \sigma(y)$ und (wegen $\Delta_w^-(y) = \left\{ \langle\rangle \mid \varphi_{\langle\rangle}(y) \right\}$) $R_w = \{ \sigma(z) \mid \varphi(\sigma(y)) \}$. Daher stimmt die Behauptung für $\sigma_w(x)$ genau mit der Definition des initialen Zustands $\sigma(x)$ überein.
2. Für ein Wort $w = \tilde{w}\langle a\rangle$ gilt:

$$\sigma_w(x) = \tau_a\big(\sigma_{\tilde{w}}(x)\big)$$

⇓ Induktionsvoraussetzung für $\sigma_{\tilde{w}}(x)$.

$$= \tau_a\big([-, z, l, R]\big) \quad \text{mit} \quad l = \sigma_{\tilde{w}}(y) \quad \text{und} \quad R = R_{\tilde{w}} = \{ \sigma_{\tilde{v}}(z) \mid \tilde{v} \in \Delta_{\tilde{w}}^-(y) \}$$

⇓ Definition des Zustandsübergangs τ_a.

$$= [-, z, l', R'] \quad \text{mit} \quad l' = \tau_a(l) \quad \text{und} \quad R' = \{ \tau_a(r) \mid r \in R \} \cup \left\{ \sigma(z) \mid \varphi(l') \right\}$$

⇓ Für l' gilt: $l' = \tau_a(l) = \tau_a\big(\sigma_{\tilde{w}}(y)\big) = \sigma_{\tilde{w}\langle a\rangle}(y) = \sigma_w(y)$.
⇓ Für $r \in R$ gilt: $r' = \tau_a(r) = \tau_a\big(\sigma_{\tilde{v}}(z)\big) = \sigma_{\tilde{v}\langle a\rangle}(z)$ mit $\tilde{v} \in \Delta_{\tilde{w}}^-(y)$.

$$= [-, z, \sigma_w(y), R'] \quad \text{mit} \quad R' = \left\{ \sigma_{\tilde{v}\langle a\rangle}(z) \,\middle|\, \tilde{v} \in \Delta_{\tilde{w}}^-(y) \right\} \cup \left\{ \sigma_{\langle\rangle}(z) \,\middle|\, \varphi_w(y) \right\}$$

⇓ Lemma.

$$= [-, z, \sigma_w(y), R'] \quad \text{mit} \quad R' = \{ \sigma_v(z) \mid v \in \Delta_w^-(y) \} = R_w$$

$$= [-, z, \sigma_w(y), R_w].$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) = \Psi(y) \cup \Phi(y)\,\Psi(z)$$

$$\Leftrightarrow \quad w \in \Psi(y) \quad \text{oder} \quad \exists\, u \in \Phi(y), v \in \Psi(z) \colon w = u\,v$$

⇓ Korrektheit des Zustandsmodells für y und z.

$$\Leftrightarrow \quad \psi_w(y) \quad \text{oder} \quad \exists\, u, v \in \Sigma^* \colon w = u\,v,\ \varphi_u(y),\ \psi_v(z)$$

⇓ Definition von $\Delta_w^-(y)$.

$$\Leftrightarrow \quad \psi_w(y) \quad \text{oder} \quad \exists\, v \in \Delta_w^-(y) \colon \psi_v(z)$$

⇓ Struktur der Folgezustände von x.

$$\Leftrightarrow \quad \psi_w(x) = \psi_w(y) \vee \bigvee_{v \in \Delta_w^-(y)} \psi_v(z) = \top,$$

und nahezu analog:

$$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top.$$

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$$\big(\sigma_w(x)\big)_q^\pi = [-, z, \sigma_w(y), R_w]_q^\pi \qquad \text{mit} \qquad R_w = \{\, \sigma_v(z) \mid v \in \Delta_w^-(y) \,\}$$

$\Downarrow$ Definition konkretisierter Zustände (§ 4.5.6.1).

$$= \Big[-, z_q^\pi, \big(\sigma_w(y)\big)_q^\pi, R'_w\Big] \qquad \text{mit} \qquad R'_w = \big(R_w\big)_q^\pi = \Big\{ \big(\sigma_v(z)\big)_q^\pi \;\Big|\; v \in \Delta_w^-(y) \Big\}$$

$\Downarrow$ Substitutionsprinzip für y und z (plus Monotonie relevanter Parameterwerte).
$\Downarrow$ Aufgrund des Substitutionsprinzips und der Definition von Δ_w^- gilt außerdem:
$\Downarrow$ $\Delta_w^-(y) = \Delta_w^-\big(y_q^\pi\big)$.

$$= \Big[-, z_q^\pi, \sigma_w\big(y_q^\pi\big), R'_w\Big] \qquad \text{mit} \qquad R'_w = \Big\{ \sigma_v\big(z_q^\pi\big) \;\Big|\; v \in \Delta_w^-\big(y_q^\pi\big) \Big\}$$

$\Downarrow$ Struktur der Folgezustände des Ausdrucks $x_q^\pi \equiv y_q^\pi - z_q^\pi$.

$$= \sigma_w\big(x_q^\pi\big).$$

B.2.3.7 Sequentielle Iteration

Gegeben sei eine sequentielle Iteration $x \equiv \ominus\, y$ mit einem beliebigen Teilausdruck y, für den das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Definition

Die Menge $\Delta_w^\ominus(y)$ der *iterativen Reste* des Wortes w bzgl. des Teilausdrucks y sei definiert als:

$$\Delta_w^\ominus(y) = \Big\{ v \in \Sigma^* \;\Big|\; \exists\, n \in \mathbb{N}_0, u_1, \ldots, u_n \in \Sigma^*\!: w = u_1 \ldots u_n\, v,\ \varphi_{u_1}(y), \ldots, \varphi_{u_n}(y) \Big\}.$$

$\Delta_w^\ominus(y)$ enthält also diejenigen Suffixe v von w, die man erhält, wenn man am Anfang von w beliebig viele vollständige Worte u_i von y entfernt.

Lemma

Für ein Wort $w = \tilde{w}\,\langle a \rangle$ gilt:

$$\Delta_w^\ominus(y) = \{\, \tilde{v}\,\langle a \rangle \mid \tilde{v} \in \Delta_{\tilde{w}}^\ominus(y) \,\} \cup \Big\{ \langle\rangle \;\Big|\; \exists\, \tilde{v} \in \Delta_{\tilde{w}}^\ominus(y)\!: \varphi_{\tilde{v}\langle a \rangle}(y) \Big\}.$$

Beweis

$$\Delta_w^\ominus(y) = \Big\{ v \in \Sigma^* \;\Big|\; \exists\, n \in \mathbb{N}_0, u_1, \ldots, u_n \in \Sigma^*\!: w = u_1 \ldots u_n\, v,\ \varphi_{u_1}(y), \ldots, \varphi_{u_n}(y) \Big\}$$

$\Downarrow$ Da w mit a endet, muß entweder v oder ein u_k mit a enden, für das $u_{k+1} = \ldots = u_n = v = \langle\rangle$
$\Downarrow$ gilt.
$\Downarrow$ Im ersten Fall gilt $\tilde{v}\,\langle a \rangle = v \in \Delta_w^\ominus(y)$ für ein Wort $\tilde{v} \in \Sigma^*$ mit $\tilde{w} = u_1 \ldots u_n\, \tilde{v}$, woraus
$\Downarrow$ $\tilde{v} \in \Delta_{\tilde{w}}^\ominus(y)$ folgt.
$\Downarrow$ Im zweiten Fall gilt offensichtlich $\langle\rangle \in \Delta_w^\ominus(y)$ und $u_k = \tilde{v}\,\langle a \rangle$ für ein Wort $\tilde{v} \in \Sigma^*$ mit
$\Downarrow$ $\tilde{w} = u_1 \ldots u_{k-1}\, \tilde{v}$, woraus ebenfalls $\tilde{v} \in \Delta_{\tilde{w}}^\ominus(y)$ und zusätzlich $\varphi_{\tilde{v}\langle a \rangle}(y) = \varphi_{u_k}(y) = \top$ folgt.

$$= \{\, \tilde{v}\,\langle a \rangle \mid \tilde{v} \in \Delta_{\tilde{w}}^\ominus(y) \,\} \cup \Big\{ \langle\rangle \;\Big|\; \exists\, \tilde{v} \in \Delta_{\tilde{w}}^\ominus(y)\!: \varphi_{\tilde{v}\langle a \rangle}(y) \Big\}.$$

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt:

$$\sigma_w(x) = [\ominus, y, T_w] \quad \text{mit} \quad T_w = \{ \sigma_w(\varepsilon) \} \cup \{ \sigma_v(y) \mid v \in \Delta_w^{\ominus}(y) \},$$

und somit:

$$\psi_w(x) = \psi_w(\varepsilon) \vee \bigvee_{v \in \Delta_w^{\ominus}(y)} \psi_v(y) \quad \text{und} \quad \varphi_w(x) = \varphi_w(\varepsilon) \vee \bigvee_{v \in \Delta_w^{\ominus}(y)} \varphi_v(y).$$

Beweis

1. Für das leere Wort $w = \langle\rangle$ gilt $\sigma_w(\varepsilon) = \top$ und $\Delta_w^{\ominus}(y) = \{ \langle\rangle \}$ und somit $T_w = \{ \top, \sigma(y) \}$. Daher stimmt die Behauptung für $\sigma_w(x)$ genau mit der Definition des initialen Zustands $\sigma(x)$ überein.
2. Für ein Wort $w = \tilde{w} \langle a \rangle$ gilt:

$$\sigma_w(x) = \tau_a\big(\sigma_{\tilde{w}}(x)\big)$$

⇓ Induktionsvoraussetzung für $\sigma_{\tilde{w}}(x)$.

$$= \tau_a\big([\ominus, y, T]\big) \quad \text{mit} \quad T = T_{\tilde{w}} = \{ \sigma_{\tilde{w}}(\varepsilon) \} \cup \{ \sigma_{\tilde{v}}(y) \mid \tilde{v} \in \Delta_{\tilde{w}}^{\ominus}(y) \}$$

⇓ Definition des Zustandsübergangs τ_a.

$$= [\ominus, y, T''] \quad \text{mit} \quad T' = \{ \tau_a(t) \mid t \in T \}$$
$$\text{und} \quad T'' = T' \cup \left\{ \sigma(y) \;\middle|\; \bigvee_{t' \in T'} \varphi(t') \right\}$$

⇓ Für $t = \sigma_{\tilde{w}}(\varepsilon) \in T$ gilt: $t' = \tau_a(t) = \tau_a\big(\sigma_{\tilde{w}}(\varepsilon)\big) = \tau_{\tilde{w}\langle a \rangle}(\varepsilon) = \tau_w(\varepsilon)$.
⇓ Für $t = \sigma_{\tilde{v}}(y) \in T$ gilt: $t' = \tau_a(t) = \tau_a\big(\sigma_{\tilde{v}}(y)\big) = \sigma_{\tilde{v}\langle a \rangle}(y)$ mit $\tilde{v} \in \Delta_{\tilde{w}}^{\ominus}(y)$.

$$= [\ominus, y, T''] \quad \text{mit} \quad T' = \{ \tau_w(\varepsilon) \} \cup \left\{ \sigma_{\tilde{v}\langle a \rangle}(y) \;\middle|\; \tilde{v} \in \Delta_{\tilde{w}}^{\ominus}(y) \right\}$$
$$\text{und} \quad T'' = T' \cup \left\{ \sigma(y) \;\middle|\; \bigvee_{t' \in T'} \varphi(t') \right\}$$

⇓ Setze T' in $\bigvee_{t' \in T'} \varphi(t')$ ein.
⇓ Beachte $\tau_w(\varepsilon) = \bot$ (da $w \neq \langle\rangle$) und somit $\varphi(t') = \bot$ für $t' = \tau_w(\varepsilon)$.

$$= [\ominus, y, T''] \quad \text{mit} \quad T' = \{ \tau_w(\varepsilon) \} \cup \left\{ \sigma_{\tilde{v}\langle a \rangle}(y) \;\middle|\; \tilde{v} \in \Delta_{\tilde{w}}^{\ominus}(y) \right\}$$
$$\text{und} \quad T'' = T' \cup \left\{ \sigma_{\langle\rangle}(y) \;\middle|\; \exists\, \tilde{v} \in \Delta_{\tilde{w}}^{\ominus}(y) : \varphi_{\tilde{v}\langle a \rangle}(y) \right\}$$

⇓ Lemma.

$$= [\ominus, y, T''] \quad \text{mit} \quad T'' = \{ \tau_w(\varepsilon) \} \cup \{ \sigma_v(y) \mid v \in \Delta_w^{\ominus}(y) \} = T_w$$
$$= [\ominus, y, T_w].$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt einerseits:

$$w \in \Psi(x) = \{ \langle\rangle \} \cup \Psi(x) = \{ \langle\rangle \} \cup \Phi(y)^* \, \Psi(y)$$

$$\Leftrightarrow \quad w = \langle\rangle \quad \text{oder} \quad \exists\, n \in \mathbb{N}_0, u_1, \ldots, u_n \in \Phi(y), v \in \Psi(y) : w = u_1 \ldots u_n v$$

⇓ Korrektheit des Zustandsmodells für y.

$$\Leftrightarrow \quad w = \langle\rangle \quad \text{oder} \quad \exists\, n \in \mathbb{N}_0, u_1, \ldots, u_n, v \in \Sigma^* : w = u_1 \ldots u_n v, \varphi_{u_1}(y), \ldots, \varphi_{u_n}(y), \psi_v(y)$$

⇓ Definition von $\Delta_w^{\ominus}(y)$.

$\Leftrightarrow \quad w = \langle\rangle \qquad \text{oder} \qquad \exists\, v \in \Delta_w^{\ominus}(y)\colon \psi_v(y)$

⇓ Struktur der Folgezustände von x.

$\Leftrightarrow \quad \psi_w(x) = \psi_w(\varepsilon) \vee \bigvee_{v \in \Delta_w^{\ominus}(y)} \psi_v(y) = \top,$

und andererseits:

$w \in \Phi(x) = \Phi(y)^*$

$\Leftrightarrow \quad w = \langle\rangle \qquad \text{oder} \qquad \exists\, n \in \mathbb{N}_0,\, u_1, \ldots, u_n,\, v \in \Phi(y)\colon w = u_1 \ldots u_n\, v$

⇓ Korrektheit des Zustandsmodells für y.

$\Leftrightarrow \quad w = \langle\rangle \qquad \text{oder} \qquad \exists\, n \in \mathbb{N}_0,\, u_1, \ldots, u_n,\, v \in \Sigma^*\colon w = u_1 \ldots u_n\, v,\ \varphi_{u_1}(y), \ldots, \varphi_{u_n}(y),\ \varphi_v(y)$

⇓ Definition von $\Delta_w^{\ominus}(y)$.

$\Leftrightarrow \quad w = \langle\rangle \qquad \text{oder} \qquad \exists\, v \in \Delta_w^{\ominus}(y)\colon \varphi_v(y)$

⇓ Struktur der Folgezustände von x.

$\Leftrightarrow \quad \varphi_w(x) = \varphi_w(\varepsilon) \vee \bigvee_{v \in \Delta_w^{\ominus}(y)} \varphi_v(y) = \top.$

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$\big(\sigma_w(x)\big)_q^\pi = [\ominus, y, T_w]_q^\pi \qquad \text{mit} \qquad T_w = \{\, \sigma_w(\varepsilon) \,\} \cup \{\, \sigma_v(y) \mid v \in \Delta_w^{\ominus}(y) \,\}$

⇓ Definition konkretisierter Zustände (§ 4.5.6.1).

$= \big[\ominus, y_q^\pi, T'_w\big] \qquad \text{mit} \qquad T'_w = (T_w)_q^\pi = \Big\{ \big(\sigma_w(\varepsilon)\big)_q^\pi \Big\} \cup \Big\{ \big(\sigma_v(y)\big)_q^\pi \;\Big|\; v \in \Delta_w^{\ominus}(y) \Big\}$

⇓ Substitutionsprinzip für y (plus Monotonie relevanter Parameterwerte).
⇓ Aufgrund des Substitutionsprinzips und der Definition von $\Delta_w^{\ominus}$ gilt außerdem:
⇓ $\Delta_w^{\ominus}(y) = \Delta_w^{\ominus}\big(y_q^\pi\big)$.
⇓ Für den leeren Ausdruck ε gilt $\varepsilon_q^\pi = \varepsilon$.

$= \big[\ominus, y_q^\pi, T'_w\big] \qquad \text{mit} \qquad T'_w = \{\, \sigma_w(\varepsilon) \,\} \cup \Big\{ \sigma_v\big(y_q^\pi\big) \;\Big|\; v \in \Delta_w^{\ominus}\big(y_q^\pi\big) \Big\}$

⇓ Struktur der Folgezustände des Ausdrucks $x_q^\pi \equiv \ominus\, y_q^\pi$.

$= \sigma_w\big(x_q^\pi\big).$

B.2.3.8 Parallele Komposition

Gegeben sei eine parallele Komposition $x \equiv y \circledcirc z$ mit beliebigen Teilausdrücken y und z, für die das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Definition

Die Menge $\Delta_w^{\circledcirc}$ der *parallelen Zerlegungen* des Wortes w sei definiert als:

$\Delta_w^{\circledcirc} = \{\, [u, v] \in \Sigma^* \times \Sigma^* \mid w \in u \otimes v \,\}.$

$\Delta_w^{\circledcirc}$ enthält also alle Wortpaare $[u, v]$, deren Verschränkung $u \otimes v$ das Wort w enthält.

Lemma

Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

$$\Delta_w^{\circledcirc} = \{\, [\tilde{u}\,\langle a\rangle, \tilde{v}] \mid [\tilde{u}, \tilde{v}] \in \Delta_{\tilde{w}}^{\circledcirc} \,\} \cup \{\, [\tilde{u}, \tilde{v}\,\langle a\rangle] \mid [\tilde{u}, \tilde{v}] \in \Delta_{\tilde{w}}^{\circledcirc} \,\}.$$

Beweis

$$\Delta_w^{\circledcirc} = \{\, [u, v] \in \Sigma^* \times \Sigma^* \mid w \in u \otimes v \,\}$$

⇓ Da w mit a endet, muß entweder u oder v mit a enden,
⇓ d. h. es gilt entweder: $u = \tilde{u}\,\langle a\rangle$ mit $\tilde{w} \in \tilde{u} \otimes v$,
⇓ oder: $v = \tilde{v}\,\langle a\rangle$ mit $\tilde{w} \in u \otimes \tilde{v}$.

$$= \{\, [\tilde{u}\,\langle a\rangle, v] \mid \tilde{w} \in \tilde{u} \otimes v \,\} \cup \{\, [u, \tilde{v}\,\langle a\rangle] \mid \tilde{w} \in u \otimes \tilde{v} \,\}$$

⇓ Definition von $\Delta_{\tilde{w}}^{\circledcirc}$.

$$= \{\, [\tilde{u}\,\langle a\rangle, v] \mid [\tilde{u}, v] \in \Delta_{\tilde{w}}^{\circledcirc} \,\} \cup \{\, [u, \tilde{v}\,\langle a\rangle] \mid [u, \tilde{v}] \in \Delta_{\tilde{w}}^{\circledcirc} \,\}.$$

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt:

$$\sigma_w(x) = [\circledcirc, A_w] \quad \text{mit} \quad A_w = \{\, [\sigma_u(y), \sigma_v(z)] \mid [u, v] \in \Delta_w^{\circledcirc} \,\},$$

und somit:

$$\psi_w(x) = \bigvee_{[u,\, v] \in \Delta_w^{\circledcirc}} \big(\psi_u(y) \wedge \psi_v(z)\big) \quad \text{und} \quad \varphi_w(x) = \bigvee_{[u,\, v] \in \Delta_w^{\circledcirc}} \big(\varphi_u(y) \wedge \varphi_v(z)\big).$$

Beweis

1. Für das leere Wort $w = \langle\rangle$ gilt (wegen $\Delta_{\langle\rangle}^{\circledcirc} = \{\, [\langle\rangle, \langle\rangle] \,\}$) $A_w = \big\{\, \big[\sigma_{\langle\rangle}(y), \sigma_{\langle\rangle}(z)\big] \,\big\} = \{\, [\sigma(y), \sigma(z)] \,\}$. Daher stimmt die Behauptung für $\sigma_w(x)$ genau mit der Definition des initialen Zustands $\sigma(x)$ überein.

2. Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

$$\sigma_w(x) = \tau_a\big(\sigma_{\tilde{w}}(x)\big)$$

⇓ Induktionsvoraussetzung für $\sigma_{\tilde{w}}(x)$.

$$= \tau_a\big([\circledcirc, A]\big) \quad \text{mit} \quad A = A_{\tilde{w}} = \{\, [\sigma_{\tilde{u}}(y), \sigma_{\tilde{v}}(z)] \mid [\tilde{u}, \tilde{v}] \in \Delta_{\tilde{w}}^{\circledcirc} \,\}$$

⇓ Definition des Zustandsübergangs τ_a.

$$= [\circledcirc, A'] \quad \text{mit} \quad A' = \{\, [\tau_a(l), r] \mid [l, r] \in A \,\} \cup \{\, [l, \tau_a(r)] \mid [l, r] \in A \,\}$$

⇓ Für $[l, r] \in A$ gilt: $[\tau_a(l), r] = \big[\tau_a\big(\sigma_{\tilde{u}}(y)\big), \sigma_{\tilde{v}}(z)\big] = \big[\sigma_{\tilde{u}\langle a\rangle}(y), \sigma_{\tilde{v}}(z)\big]$
⇓ und: $[l, \tau_a(r)] = \big[\sigma_{\tilde{u}}(y), \tau_a\big(\sigma_{\tilde{v}}(z)\big)\big] = \big[\sigma_{\tilde{u}}(y), \sigma_{\tilde{v}\langle a\rangle}(z)\big]$,
⇓ jeweils mit $[\tilde{u}, \tilde{v}] \in \Delta_{\tilde{w}}^{\circledcirc}$.

$$= [\circledcirc, A'] \quad \text{mit} \quad A' = \big\{\, \big[\sigma_{\tilde{u}\langle a\rangle}(y), \sigma_{\tilde{v}}(z)\big] \,\big|\, [\tilde{u}, \tilde{v}] \in \Delta_{\tilde{w}}^{\circledcirc} \,\big\} \cup \big\{\, \big[\sigma_{\tilde{u}}(y), \sigma_{\tilde{v}\langle a\rangle}(z)\big] \,\big|\, [\tilde{u}, \tilde{v}] \in \Delta_{\tilde{w}}^{\circledcirc} \,\big\}$$

⇓ Lemma.

$$= [\circledcirc, A'] \quad \text{mit} \quad A' = \{\, [\sigma_u(y), \sigma_v(z)] \mid [u, v] \in \Delta_w^{\circledcirc} \,\} = A_w$$

$$= [\circledcirc, A_w].$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) = \Psi(y) \otimes \Psi(z)$$

$$\Leftrightarrow \quad \exists\, u \in \Psi(y),\ v \in \Psi(z) \colon w \in u \otimes v$$

⇓ Korrektheit des Zustandsmodells für y und z.

$$\Leftrightarrow \quad \exists\, u, v \in \Sigma^* \colon w \in u \otimes v,\ \psi_u(y),\ \psi_v(z)$$

⇓ Definition von $\Delta_w^{\circledcirc}$.

$$\Leftrightarrow \quad \exists\, [u, v] \in \Delta_w^{\circledcirc} \colon \psi_u(y),\ \psi_v(z)$$

⇓ Struktur der Folgezustände von x.

$$\Leftrightarrow \quad \psi_w(x) = \bigvee_{[u,\, v] \in \Delta_w^{\circledcirc}} \big(\psi_u(y) \wedge \psi_v(z)\big) = \top,$$

und analog:

$$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top.$$

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$$\big(\sigma_w(x)\big)_q^\pi = [\circledcirc, A_w]_q^\pi \qquad \text{mit} \quad A_w = \{\, [\sigma_u(y), \sigma_v(z)] \mid [u, v] \in \Delta_w^{\circledcirc} \,\}$$

⇓ Definition konkretisierter Zustände (§ 4.5.6.1).

$$= [\circledcirc, A'_w] \qquad \text{mit} \quad A'_w = \big(A_w\big)_q^\pi = \Big\{ \Big[\big(\sigma_u(y)\big)_q^\pi, \big(\sigma_v(z)\big)_q^\pi\Big] \,\Big|\, [u, v] \in \Delta_w^{\circledcirc} \Big\}$$

⇓ Substitutionsprinzip für y und z (plus Monotonie relevanter Parameterwerte).

$$= [\circledcirc, A'_w] \qquad \text{mit} \quad A'_w = \Big\{ \Big[\sigma_u\big(y_q^\pi\big), \sigma_v\big(z_q^\pi\big)\Big] \,\Big|\, [u, v] \in \Delta_w^{\circledcirc} \Big\}$$

⇓ Struktur der Folgezustände des Ausdrucks $x_q^\pi \equiv y_q^\pi \circledcirc z_q^\pi$.

$$= \sigma_w\big(x_q^\pi\big).$$

B.2.4 Beweis für Quantorausdrücke

B.2.4.1 Disjunktions-Quantorausdrücke

Gegeben sei ein Disjunktions-Quantorausdruck $x \equiv \bigcirc_p\, y$ mit einem beliebigen Teilausdruck y, für den und dessen Konkretisierungen y_p^ω das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt:

$$\sigma_w(x) = [\bigcirc, y, p, \sigma_w(y), T_w] \qquad \text{mit} \quad T_w = \Big\{ \Big[\sigma_w\big(y_p^\omega\big), \omega\Big] \,\Big|\, \omega \in \Omega_w(y, p) \Big\},$$

und somit:

$$\psi_w(x) = \psi_w(y) \vee \bigvee_{\omega \in \Omega_w(y,\, p)} \psi_w\big(y_p^\omega\big) \qquad \text{und} \qquad \varphi_w(x) = \varphi_w(y) \vee \bigvee_{\omega \in \Omega_w(y,\, p)} \varphi_w\big(y_p^\omega\big).$$

Beweis

1. Für das leere Wort $w = \langle\rangle$ gilt $\sigma_w(y) = \sigma(y)$ und (wegen $\Omega_w(y, p) = \emptyset$) $T_w = \emptyset$. Daher stimmt die Behauptung für $\sigma_w(x)$ genau mit der Definition des initialen Zustands $\sigma(x)$ überein.
2. Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

$$\sigma_w(x) = \tau_a\big(\sigma_{\tilde{w}}(x)\big)$$

⇓ Induktionsvoraussetzung für $\sigma_{\tilde{w}}(x)$.

$$= \tau_a\big([\bigcirc, y, p, \sigma_{\tilde{w}}(y), T]\big) \quad \text{mit} \quad T = T_{\tilde{w}} = \left\{ \left[\sigma_{\tilde{w}}\big(y_p^\omega\big), \omega\right] \;\middle|\; \omega \in \Omega_{\tilde{w}}(y, p) \right\}$$

⇓ Definition des Zustandsübergangs τ_a.

$$= \left[\bigcirc, y, p, \tau_a\big(\sigma_{\tilde{w}}(y)\big), T''\right] \quad \text{mit} \quad T' = T \cup \left\{ \left[\big(\sigma_{\tilde{w}}(y)\big)_p^\omega, \omega\right] \;\middle|\; \omega \in \Omega_a(y, p) \setminus \Omega(T) \right\}$$
$$\text{und} \quad T'' = \{ \tau_a(t) \mid t \in T' \}$$

⇓ Substitutionsprinzip für y (beachte $\omega \notin \Omega(T) = \Omega_{\tilde{w}}(y, p)$).

$$= [\bigcirc, y, p, \sigma_w(y), T''] \quad \text{mit} \quad T' = T \cup \left\{ \left[\sigma_{\tilde{w}}\big(y_p^\omega\big), \omega\right] \;\middle|\; \omega \in \Omega_a(y, p) \setminus \Omega_{\tilde{w}}(y, p) \right\}$$
$$\text{und} \quad T'' = \{ \tau_a(t) \mid t \in T' \}$$

⇓ Setze $T = T_{\tilde{w}}$ ein und beachte $\Omega_{\tilde{w}}(y, p) \cup \big(\Omega_a(y, p) \setminus \Omega_{\tilde{w}}(y, p)\big) = \Omega_w(y, p)$.

$$= [\bigcirc, y, p, \sigma_w(y), T''] \quad \text{mit} \quad T' = \left\{ \left[\sigma_{\tilde{w}}\big(y_p^\omega\big), \omega\right] \;\middle|\; \omega \in \Omega_w(y, p) \right\}$$
$$\text{und} \quad T'' = \{ \tau_a(t) \mid t \in T' \}$$

⇓ Definition des Zustandsübergangs τ_a für erweiterte Zustände $t \in T'$.

$$= [\bigcirc, y, p, \sigma_w(y), T''] \quad \text{mit} \quad T'' = \left\{ \left[\sigma_w\big(y_p^\omega\big), \omega\right] \;\middle|\; \omega \in \Omega_w(y, p) \right\} = T_w$$
$$= [\bigcirc, y, p, \sigma_w(y), T_w].$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) = \bigcup_{\omega\in\Omega} \Psi\big(y_p^\omega\big)$$

$$\Leftrightarrow \quad \exists\, \omega \in \Omega\colon w \in \Psi\big(y_p^\omega\big)$$

⇓ Korrektheit des Zustandsmodells für y_p^ω.

$$\Leftrightarrow \quad \exists\, \omega \in \Omega\colon \psi_w\big(y_p^\omega\big)$$

⇓ Unterscheide zwischen relevanten und irrelevanten Werten $\omega \in \Omega$.

$$\Leftrightarrow \quad \exists\, \omega \in \Omega_w(y, p)\colon \psi_w\big(y_p^\omega\big) \quad \text{oder} \quad \exists\, \omega \notin \Omega_w(y, p)\colon \psi_w\big(y_p^\omega\big)$$

⇓ Substitutionsprinzip für y.

$$\Leftrightarrow \quad \exists\, \omega \in \Omega_w(y, p)\colon \psi_w\big(y_p^\omega\big) \quad \text{oder} \quad \psi_w(y)$$

⇓ Struktur der Folgezustände von x.

$$\Leftrightarrow \quad \psi_w(x) = \psi_w(y) \vee \bigvee_{\omega\in\Omega_w(y,\, p)} \psi_w\big(y_p^\omega\big) = \top,$$

und analog:

$$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top.$$

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$$\big(\sigma_w(x)\big)^\pi_q = [\bigcirc, y, p, \sigma_w(y), T_w]^\pi_q \quad \text{mit} \quad T_w = \big\{ \big[\sigma_w\big(y^\omega_p\big), \omega\big] \;\big|\; \omega \in \Omega_w(y, p) \big\}$$

$$= \big[\bigcirc, y^\pi_q, p, \big(\sigma_w(y)\big)^\pi_q, T'_w\big] \quad \text{mit} \quad T'_w = \big(T_w\big)^\pi_q = \Big\{ \Big[\big(\sigma_w\big(y^\omega_p\big)\big)^\pi_q, \omega\Big] \;\Big|\; \omega \in \Omega_w(y, p) \Big\}$$

⇓ Substitutionsprinzip für y bzw. y^ω_p bzgl. des Parameters q.

$$= \big[\bigcirc, y^\pi_q, p, \sigma_w\big(y^\pi_q\big), T'_w\big] \quad \text{mit} \quad T'_w = \big\{ \big[\sigma_w\big(y^{\omega,\,\pi}_{p,\,q}\big), \omega\big] \;\big|\; \omega \in \Omega_w(y, p) \big\}$$

⇓ Invarianz relevanter Parameterwerte (§ B.2.1.2).

$$= \big[\bigcirc, y^\pi_q, p, \sigma_w\big(y^\pi_q\big), T'_w\big] \quad \text{mit} \quad T'_w = \big\{ \big[\sigma_w\big(y^{\pi,\,\omega}_{q,\,p}\big), \omega\big] \;\big|\; \omega \in \Omega_w\big(y^\pi_q, p\big) \big\}$$

⇓ Ersetze y^π_q durch $\tilde{y}$.

$$= [\bigcirc, \tilde{y}, p, \sigma_w(\tilde{y}), T'_w] \quad \text{mit} \quad T'_w = \big\{ \big[\sigma_w\big(\tilde{y}^\omega_p\big), \omega\big] \;\big|\; \omega \in \Omega_w(\tilde{y}, p) \big\}$$

⇓ Struktur der Folgezustände des Ausdrucks $\tilde{x} \equiv \underset{p}{\bigcirc}\, \tilde{y} \equiv \underset{p}{\bigcirc}\, y^\pi_q \equiv x^\pi_q$.

$$= \sigma_w(\tilde{x}) = \sigma_w\big(x^\pi_q\big).$$

B.2.4.2 Konjunktions-Quantorausdrücke

Da das Zustandsmodell eines Konjunktions-Quantorausdrucks $x \equiv \underset{p}{\bullet}\, y$ vollkommen analog bzw. dual zu dem eines Disjunktions-Quantorausdrucks definiert ist, ergibt sich auch die Struktur der Folgezustände von x sowie die Korrektheit des Zustandsmodells und die Gültigkeit des Substitutionsprinzips analog.

B.2.4.3 Synchronisations-Quantorausdrücke

Gegeben sei ein Synchronisations-Quantorausdruck $x \equiv \underset{p}{\circledcirc}\, y$ mit einem beliebigen Teilausdruck y, für den und dessen Konkretisierungen y^ω_p das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt:

$$\sigma_w(x) = \big[\circledcirc, y, p, \sigma_{w_p}(y), T_w\big] \quad \text{mit} \quad T_w = \big\{ \big[\sigma_{w_\omega}\big(y^\omega_p\big), \omega\big] \;\big|\; \omega \in \Omega_w(y, p) \big\}$$
$$\text{und} \quad w_\omega = w/\kappa_x\big(y^\omega_p\big) \quad \text{für} \quad \omega \in \Omega \cup \{ p \},$$

und somit:

$$\psi_w(x) = \psi_{w_p}(y) \wedge \bigwedge_{\omega \in \Omega_w(y,\, p)} \psi_{w_\omega}\big(y^\omega_p\big) \quad \text{und} \quad \varphi_w(x) = \varphi_{w_p}(y) \wedge \bigwedge_{\omega \in \Omega_w(y,\, p)} \varphi_{w_\omega}\big(y^\omega_p\big).$$

Beweis

1. Für das leere Wort $w = \langle\rangle$ gilt $\sigma_{w_p}(y) = \sigma(y)$ und (wegen $\Omega_w(y, p) = \emptyset$) $T_w = \emptyset$. Daher stimmt die Behauptung für $\sigma_w(x)$ genau mit der Definition des initialen Zustands $\sigma(x)$ überein.

2. Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

$$\sigma_w(x) = \tau_a\big(\sigma_{\tilde{w}}(x)\big)$$

$\Downarrow$ Induktionsvoraussetzung für $\sigma_{\tilde{w}}(x)$.

$$= \tau_a\Big(\big[\boldsymbol{\circ}, y, p, \sigma_{\tilde{w}_p}(y), T\big]\Big) \qquad \text{mit} \quad T = T_{\tilde{w}} = \Big\{\big[\sigma_{\tilde{w}_\omega}\big(y_p^\omega\big), \omega\big] \;\Big|\; \omega \in \Omega_{\tilde{w}}(y, p)\Big\}$$
$$\text{und} \quad \tilde{w}_\omega = \tilde{w}/\kappa_x\big(y_p^\omega\big) \quad \text{für} \quad \omega \in \Omega \cup \{\, p\,\}$$

$\Downarrow$ Definition des Zustandsübergangs τ_a.

$$= \Big[\boldsymbol{\circ}, y, p, \vartheta_a^p\big(\sigma_{\tilde{w}_p}(y)\big), T''\Big] \qquad \text{mit} \quad T' = T \cup \Big\{\Big[\big(\sigma_{\tilde{w}_p}(y)\big)_p^\omega, \omega\Big] \;\Big|\; \omega \in \Omega_a(y, p) \setminus \Omega(T)\Big\}$$
$$\text{und} \quad T'' = \{\, \vartheta_a^\omega(t) \mid t \in T'\,\}$$
$$\text{sowie} \quad \vartheta_a^\omega(t) = \begin{cases} t & \text{für} \quad a \in \Big(\bigcup\limits_{\pi\in\Omega} (\alpha(y))_p^\pi\Big) \setminus (\alpha(y))_p^\omega = \alpha(x) \setminus \alpha\big(y_p^\omega\big) = \kappa_x\big(y_p^\omega\big) \\ \tau_a(t) & \text{sonst} \end{cases}$$

$\Downarrow$ Für $\omega \in \Omega \cup \{\, p\,\}$ gilt:

$\Downarrow$ $w_\omega = \big(\tilde{w}\,\langle a\rangle\big)/\kappa_x\big(y_p^\omega\big) = \tilde{w}_\omega\,\big(\langle a\rangle/\kappa_x\big(y_p^\omega\big)\big) = \begin{cases} \tilde{w}_\omega & \text{für } a \in \kappa_x\big(y_p^\omega\big), \\ \tilde{w}_\omega\,\langle a\rangle & \text{sonst.} \end{cases}$

$\Downarrow$ Daher gilt für $t = \sigma_{\tilde{w}_\omega}\big(y_p^\omega\big)$:

$\Downarrow$ $\vartheta_a^\omega(t) = \begin{cases} t = \sigma_{\tilde{w}_\omega}\big(y_p^\omega\big) = \sigma_{w_\omega}\big(y_p^\omega\big) & \text{für } a \in \kappa_x\big(y_p^\omega\big), \\ \tau_a(t) = \tau_a\big(\sigma_{\tilde{w}_\omega}\big(y_p^\omega\big)\big) = \sigma_{\tilde{w}_\omega\langle a\rangle}\big(y_p^\omega\big) = \sigma_{w_\omega}\big(y_p^\omega\big) & \text{sonst,} \end{cases}$

$\Downarrow$ d. h. es gilt generell $\vartheta_a^\omega\big(\sigma_{\tilde{w}_\omega}\big(y_p^\omega\big)\big) = \sigma_{w_\omega}\big(y_p^\omega\big)$ für $\omega \in \Omega \cup \{\, p\,\}$.

$$= \big[\boldsymbol{\circ}, y, p, \sigma_{w_p}(y), T''\big] \qquad \text{mit} \quad T' \text{ und } T'' \text{ wie oben}$$

$\Downarrow$ Ähnlich wie bei der elementaren Synchronisation (§ B.2.3.5, Gültigkeit des Substitu-
$\Downarrow$ tionsprinzips) gilt für $\omega \notin \Omega_{\tilde{w}}(y, p)$: $\tilde{w}_p = \tilde{w}/\kappa_x(y) = \tilde{w}/\kappa_x\big(y_p^\omega\big) = \tilde{w}_\omega$,
$\Downarrow$ und somit: $\sigma_{\tilde{w}_p}(y) = \sigma_{\tilde{w}_\omega}(y)$ für $\omega \notin \Omega(T) = \Omega_{\tilde{w}}(y, p)$.
$\Downarrow$ Aufgrund des Substitutionsprinzips gilt außerdem: $\big(\sigma_{\tilde{w}_\omega}(y)\big)_p^\omega = \sigma_{\tilde{w}_\omega}\big(y_p^\omega\big)$.

$$= \big[\boldsymbol{\circ}, y, p, \sigma_{w_p}(y), T''\big] \qquad \text{mit} \quad T' = T \cup \Big\{\big[\sigma_{\tilde{w}_\omega}\big(y_p^\omega\big), \omega\big] \;\Big|\; \omega \in \Omega_a(y, p) \setminus \Omega_{\tilde{w}}(y, p)\Big\}$$
$$\text{und} \quad T'' = \{\, \vartheta_a^\omega(t) \mid t \in T'\,\}$$

$\Downarrow$ Setze $T = T_{\tilde{w}}$ ein und beachte $\Omega_{\tilde{w}}(y, p) \cup \big(\Omega_a(y, p) \setminus \Omega_{\tilde{w}}(y, p)\big) = \Omega_w(y, p)$.

$$= \big[\boldsymbol{\circ}, y, p, \sigma_{w_p}(y), T''\big] \qquad \text{mit} \quad T' = \Big\{\big[\sigma_{\tilde{w}_\omega}\big(y_p^\omega\big), \omega\big] \;\Big|\; \omega \in \Omega_w(y, p)\Big\}$$
$$\text{und} \quad T'' = \{\, \vartheta_a^\omega(t) \mid t \in T'\,\}$$

$\Downarrow$ $\vartheta_a^\omega\big(\sigma_{\tilde{w}_\omega}\big(y_p^\omega\big)\big) = \sigma_{w_\omega}\big(y_p^\omega\big)$ (siehe oben).

$$= \big[\boldsymbol{\circ}, y, p, \sigma_{w_p}(y), T''\big] \qquad \text{mit} \quad T'' = \Big\{\big[\sigma_{w_\omega}\big(y_p^\omega\big), \omega\big] \;\Big|\; \omega \in \Omega_w(y, p)\Big\} = T_w$$

$$= \big[\boldsymbol{\circ}, y, p, \sigma_{w_p}(y), T_w\big].$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) = \bigcap_{\omega \in \Omega} \Psi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$$

$$\Leftrightarrow \quad \forall\, \omega \in \Omega\colon w \in \Psi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$$

$$\Leftrightarrow \quad \forall\, \omega \in \Omega\ \exists\, u_\omega \in \Psi\left(y_p^\omega\right),\, v_\omega \in \kappa_x\left(y_p^\omega\right)^*\colon w \in u_\omega \otimes v_\omega$$

$\Downarrow$ Aus $w \in u_\omega \otimes v_\omega$ mit $u_\omega \in \Psi\left(y_p^\omega\right) \subseteq \alpha\left(y_p^\omega\right)^* =: U^*$ und $v_\omega \in \kappa_x\left(y_p^\omega\right)^* =: V^*$ folgt wegen
$\Downarrow$ $U \cap V = \varnothing$ (vgl. Lemma in § B.2.3.5):
$\Downarrow$ $u_\omega = w/V = w/\kappa_x\left(y_p^\omega\right) = w_\omega$.

$$\Leftrightarrow \quad \forall\, \omega \in \Omega\colon w_\omega = w/\kappa_x\left(y_p^\omega\right) \in \Psi\left(y_p^\omega\right)$$

$\Downarrow$ Korrektheit des Zustandsmodells für y_p^ω.

$$\Leftrightarrow \quad \forall\, \omega \in \Omega\colon \psi_{w_\omega}\left(y_p^\omega\right)$$

$\Downarrow$ Unterscheide zwischen relevanten und irrelevanten Werten $\omega \in \Omega$.

$$\Leftrightarrow \quad \forall\, \omega \in \Omega_w(y,\, p)\colon \psi_{w_\omega}\left(y_p^\omega\right) \quad \text{und} \quad \forall\, \omega \notin \Omega_w(y,\, p)\colon \psi_{w_\omega}\left(y_p^\omega\right)$$

$\Downarrow$ Substitutionsprinzip für y sowie $\sigma_{w_\omega}(y) = \sigma_{w_p}(y)$ für $\omega \notin \Omega_w(y,\, p)$ (siehe oben).

$$\Leftrightarrow \quad \forall\, \omega \in \Omega_w(y,\, p)\colon \psi_{w_\omega}\left(y_p^\omega\right) \quad \text{und} \quad \psi_{w_p}(y)$$

$\Downarrow$ Struktur der Folgezustände von x.

$$\Leftrightarrow \quad \psi_w(x) = \psi_{w_p}(y) \wedge \bigwedge_{\omega \in \Omega_w(y,\, p)} \psi_{w_\omega}\left(y_p^\omega\right) = \top,$$

und analog:

$$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top.$$

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$$\left(\sigma_w(x)\right)_q^\pi = \left[\mathbf{o},\, y,\, p,\, \sigma_{w_p}(y),\, T_w\right]_q^\pi \qquad \text{mit} \quad T_w = \left\{ \left[\sigma_{w_\omega}\left(y_p^\omega\right),\, \omega\right] \,\middle|\, \omega \in \Omega_w(y,\, p) \right\}$$

$$= \left[\mathbf{o},\, y_q^\pi,\, p,\, \left(\sigma_{w_p}(y)\right)_q^\pi,\, T_w'\right] \qquad \text{mit} \quad T_w' = \left(T_w\right)_q^\pi = \left\{ \left[\left(\sigma_{w_\omega}\left(y_p^\omega\right)\right)_q^\pi,\, \omega\right] \,\middle|\, \omega \in \Omega_w(y,\, p) \right\}$$

$\Downarrow$ Substitutionsprinzip für y bzw. y_p^ω bzgl. des Parameters q.

$$= \left[\mathbf{o},\, y_q^\pi,\, p,\, \sigma_{w_p}\left(y_q^\pi\right),\, T_w'\right] \qquad \text{mit} \quad T_w' = \left\{ \left[\sigma_{w_\omega}\left(y_{p,\,q}^{\omega,\,\pi}\right),\, \omega\right] \,\middle|\, \omega \in \Omega_w(y,\, p) \right\}$$

$\Downarrow$ Invarianz relevanter Parameterwerte (§ B.2.1.2).
$\Downarrow$ Außerdem gilt analog zur elementaren Synchronisation:
$\Downarrow$ $w_\omega = w/\kappa_x\left(y_p^\omega\right) = w/\kappa_{\tilde{x}}\left(\tilde{y}_p^\omega\right) =: \tilde{w}_\omega$ für $\omega \in \Omega \cup \{\, p \,\}$,
$\Downarrow$ wenn man $\tilde{x} \equiv x_q^\pi \equiv \underset{p}{\mathbf{O}}\, y_q^\pi \equiv \underset{p}{\mathbf{O}}\, \tilde{y}$ setzt.

$$= \left[\mathbf{o},\, \tilde{y},\, p,\, \sigma_{\tilde{w}_p}(\tilde{y}),\, T_w'\right] \qquad \text{mit} \quad T_w' = \left\{ \left[\sigma_{\tilde{w}_\omega}\left(\tilde{y}_p^\omega\right),\, \omega\right] \,\middle|\, \omega \in \Omega_w(\tilde{y},\, p) \right\}$$

$\Downarrow$ Struktur der Folgezustände des Ausdrucks $\tilde{x}$.

$$= \sigma_w(\tilde{x}) = \sigma_w\left(x_q^\pi\right).$$

B.2.4.4 Parallele Quantorausdrücke

Gegeben sei ein paralleler Quantorausdruck $x \equiv \underset{p}{\circledcirc}\, y$ mit einem beliebigen Teilausdruck y, für den und dessen Konkretisierungen y_p^{ω} das Korrektheitstheorem gemäß der globalen Induktionsvoraussetzung bereits bewiesen sei.

Definition

Die Menge $\Delta_w^{\circledcirc}$ der *unendlichen parallelen Zerlegungen* des Wortes w sei definiert als:

$$\Delta_w^{\circledcirc} = \left\{ \sum_{i=1}^{\infty} u_i \;\middle|\; \exists\, n \in \mathbb{N}_0 \colon w \in \bigotimes_{i=1}^{n} u_i,\ u_i = \langle\rangle \text{ für } i > n \right\}.$$

$\Delta_w^{\circledcirc}$ enthält also alle Multimengen $\sum_{i=1}^{\infty} u_i$ von Worten u_i, deren Verschränkung das Wort w enthält. Da w endlich ist, sind jeweils fast alle u_i leer.

Lemma

Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

$$\Delta_w^{\circledcirc} = \left\{ \sum_{i=1}^{\infty} \tilde{u}_i - \tilde{u}_k + \tilde{u}_k\,\langle a\rangle \;\middle|\; \sum_{i=1}^{\infty} \tilde{u}_i \in \Delta_{\tilde{w}}^{\circledcirc},\ k \in \mathbb{N} \right\}.$$

Beweis

Durch geeignete Verallgemeinerung des Lemma-Beweises in § B.2.3.8.

Struktur der Folgezustände

Für die Folgezustände des Ausdrucks x gilt:

$$\sigma_w(x) = [\circledcirc,\, y,\, p,\, A_w] \quad \text{mit} \quad A_w = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{u_i}\!\left(y_p^{\omega_i}\right),\, \omega_i\right] \;\middle|\; \sum_{i=1}^{\infty} u_i \in \Delta_w^{\circledcirc},\ \sum_{i=1}^{\infty} \omega_i = \Omega_w(y,\, p) + \sum_{1}^{\infty} p \right\},$$

und somit:

$$\psi_w(x) = \bigvee_{T \in A_w} \bigwedge_{t \in T} \psi(t) \qquad \text{und} \qquad \varphi_w(x) = \bigvee_{T \in A_w} \bigwedge_{t \in T} \varphi(t).$$

Beweis

1. Für das leere Wort $w = \langle\rangle$ gilt wegen $\Delta_w^{\circledcirc} = \left\{ \sum_{i=1}^{\infty} \langle\rangle \right\}$ und $\Omega_w(y,\, p) = \emptyset$:

   $$A_w = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{\langle\rangle}\!\left(y_p^p\right),\, p\right] \right\} = \left\{ \sum_{i=1}^{\infty} [\sigma(y),\, p] \right\}.$$

   Daher stimmt die Behauptung für $\sigma_w(x)$ genau mit der Definition des initialen Zustands $\sigma(x)$ überein.

2. Für ein Wort $w = \tilde{w}\,\langle a\rangle$ gilt:

   $\sigma_w(x) = \tau_a\big(\sigma_{\tilde{w}}(x)\big)$

   ⇓ Induktionsvoraussetzung für $\sigma_{\tilde{w}}(x)$.

$$= \tau_a([\circledcirc, y, p, A]) \quad \text{mit} \quad A = A_{\tilde{w}} = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{\tilde{u}_i}\left(y_p^{\omega_i}\right), \omega_i\right] \,\middle|\, \sum_{i=1}^{\infty} \tilde{u}_i \in \Delta_{\tilde{w}}^{\circledcirc}, \sum_{i=1}^{\infty} \omega_i = \Omega_{\tilde{w}}(y, p) + \sum_1^{\infty} p \right\}$$

⇓ Definition des Zustandsübergangs τ_a.

$$= [\circledcirc, y, p, A'] \quad \text{mit} \quad A' = \{ S - t + \tau_a(t) \mid t \in S, S \in \vartheta(T), T \in A \} \quad \text{und}$$
$$\vartheta(T) = \left\{ T - \sum_{i=1}^{k} [t_i, p] + \sum_{i=1}^{k} \left[(t_i)_p^{\omega_i}, \omega_i\right] \,\middle|\, \sum_{i=1}^{k} [t_i, p] \subseteq T, \sum_{i=1}^{k} \omega_i = \Omega_a(y, p) \setminus \Omega(T) \right\}$$

⇓ Für $T = \sum_{i=1}^{\infty} \left[\sigma_{\tilde{u}_i}\left(y_p^{\omega_i}\right), \omega_i\right] \in A$ und $[t, p] \in T$ gilt: $t = \sigma_{\tilde{u}_i}\left(y_p^{p}\right) = \sigma_{\tilde{u}_i}(y)$ für ein $i \in \mathbb{N}$.

⇓ Daher gilt aufgrund des Substitutionsprinzips für einen Wert $\omega \notin \Omega_{\tilde{w}}(y, p) = \Omega(T)$:

⇓ $t_p^{\omega} = \left(\sigma_{\tilde{u}_i}(y)\right)_p^{\omega} = \sigma_{\tilde{u}_i}\left(y_p^{\omega}\right)$.

⇓ Daraus folgt: $\vartheta(T) = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{\tilde{u}_i}\left(y_p^{\omega_i}\right), \omega_i\right] \,\middle|\, \sum_{i=1}^{\infty} \tilde{u}_i \in \Delta_{\tilde{w}}^{\circledcirc}, \sum_{i=1}^{\infty} \omega_i = \Omega_w(y, p) + \sum_1^{\infty} p \right\}$.

⇓ Für $S \in \vartheta(T)$ und $t \in S$ gilt daher:

⇓ $S - t + \tau_a(t) = \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(y_p^{\omega_i}\right), \omega_i\right] \quad \text{mit} \quad \sum_{i=1}^{\infty} u_i = \sum_{i=1}^{\infty} \tilde{u}_i - \tilde{u}_k + \tilde{u}_k \langle a \rangle$ für ein $k \in \mathbb{N}$.

$$= [\circledcirc, y, p, A'] \quad \text{mit} \quad A' = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(y_p^{\omega_i}\right), \omega_i\right] \,\middle|\, \sum_{i=1}^{\infty} u_i = \sum_{i=1}^{\infty} \tilde{u}_i - \tilde{u}_k + \tilde{u}_k \langle a \rangle, \sum_{i=1}^{\infty} \tilde{u}_i \in \Delta_{\tilde{w}}^{\circledcirc}, k \in \mathbb{N}, \sum_{i=1}^{\infty} \omega_i = \Omega_w(y, p) + \sum_1^{\infty} p \right\}$$

⇓ Lemma.

$$= [\circledcirc, y, p, A'] \quad \text{mit} \quad A' = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(y_p^{\omega_i}\right), \omega_i\right] \,\middle|\, \sum_{i=1}^{\infty} u_i \in \Delta_{w}^{\circledcirc}, \sum_{i=1}^{\infty} \omega_i = \Omega_w(y, p) + \sum_1^{\infty} p \right\} = A_w$$

$$= [\circledcirc, y, p, A_w].$$

Korrektheit des Zustandsmodells

Für ein Wort $w \in \Sigma^*$ gilt:

$$w \in \Psi(x) = \bigotimes_{\omega \in \Omega} \Psi\left(y_p^{\omega}\right)$$

⇓ Wegen $\langle\rangle \in \Psi\left(y_p^{\omega}\right)$ für alle $\omega \in \Omega$ gilt: $\bigotimes_{\omega \in \Omega} \ldots = \bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} \ldots$

$$\Leftrightarrow \quad w \in \bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \ldots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} \Psi\left(y_p^{\omega_i}\right)$$

$$\Leftrightarrow \quad \exists\, n \in \mathbb{N}, \omega_1 \neq \ldots \neq \omega_n \in \Omega, u_i \in \Psi\left(y_p^{\omega_i}\right): w \in \bigotimes_{i=1}^{n} u_i$$

⇓ Korrektheit des Zustandsmodells für $y_p^{\omega_i}$.

$$\Leftrightarrow \quad \exists\, n \in \mathbb{N}, \omega_1 \neq \ldots \neq \omega_n \in \Omega, u_i \in \Sigma^*: w \in \bigotimes_{i=1}^{n} u_i, \psi_{u_i}\left(y_p^{\omega_i}\right)$$

⇓ O. B. d. A. gilt $\{ \omega_1, \ldots, \omega_n \} \supseteq \Omega_w(y, p)$.

⇓ (Andernfalls füge fehlende Werte ω_i und zugehörige Worte $u_i = \langle\rangle$ hinzu.)

⇓ Wähle dann $u_i = \langle\rangle$ für $i > n$ und $\omega_i' = \begin{cases} \omega_i, & \text{falls } \omega_i \in \Omega_w(y, p), \\ p & \text{sonst,} \end{cases}$

⇓ und beachte das Substitutionsprinzip für y.

$\Leftrightarrow \quad \exists \sum_{i=1}^{\infty} u_i \in \Delta_w^{\circledcirc}, \sum_{i=1}^{\infty} \omega_i' = \Omega_w(y, p) + \sum_1^{\infty} p : \psi_{u_i}\left(y_p^{\omega_i}\right)$

⇓ Wähle $T = \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(y_p^{\omega_i'}\right), \omega_i'\right]$.

$\Leftrightarrow \quad \exists T \in A_w : \forall t \in T : \psi(t)$

⇓ Struktur der Folgezustände von x.

$\Leftrightarrow \quad \psi_w(x) = \bigvee_{T \in A_w} \bigwedge_{t \in T} \psi(t) = \top$.

Die Äquivalenz

$w \in \Phi(x) \quad \Leftrightarrow \quad \varphi_w(x) = \top$

ergibt sich vollkommen analog, wenn man zusätzlich die folgenden Implikationen beachtet:

$w \in \Phi(x)$

$\Rightarrow \quad \Phi(x) \neq \varnothing$

⇓ Siehe § 3.4.6.3.

$\Rightarrow \quad y$ ist singulär

⇓ Siehe § 3.4.6.2.

$\Rightarrow \quad \langle\rangle \in \Phi\left(y_p^{\omega}\right)$ für alle $\omega \in \Omega$.

Gültigkeit des Substitutionsprinzips

Für ein Wort $w \in \Sigma^*$, einen Quantorparameter $q \in \Pi$ und einen bzgl. x und q irrelevanten Parameterwert $\pi \notin \Omega_w(x, q)$ des Wortes w gilt:

$\left(\sigma_w(x)\right)_q^{\pi} = [\circledcirc, y, p, A_w]_q^{\pi}$ mit $A_w = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(y_p^{\omega_i}\right), \omega_i\right] \;\middle|\; \textit{siehe oben} \right\}$

$= \left[\circledcirc, y_q^{\pi}, p, A_w'\right]$ mit $A_w' = \left(A_w\right)_q^{\pi} = \left\{ \sum_{i=1}^{\infty} \left[\left(\sigma_{u_i}\left(y_p^{\omega_i}\right)\right)_q^{\pi}, \omega_i\right] \;\middle|\; \textit{siehe oben} \right\}$

⇓ Substitutionsprinzip für $y_p^{\omega_i}$ bzgl. des Parameters q.

$= \left[\circledcirc, y_q^{\pi}, p, A_w'\right]$ mit $A_w' = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(y_{p,q}^{\omega_i, \pi}\right), \omega_i\right] \;\middle|\; \sum_{i=1}^{\infty} u_i \in \Delta_w^{\circledcirc}, \sum_{i=1}^{\infty} \omega_i = \Omega_w(y, p) + \sum_1^{\infty} p \right\}$

⇓ Invarianz relevanter Parameterwerte (§ B.2.1.2).

$= \left[\circledcirc, y_q^{\pi}, p, A_w'\right]$ mit $A_w' = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(y_{q,p}^{\pi, \omega_i}\right), \omega_i\right] \;\middle|\; \sum_{i=1}^{\infty} u_i \in \Delta_w^{\circledcirc}, \sum_{i=1}^{\infty} \omega_i = \Omega_w\left(y_q^{\pi}, p\right) + \sum_1^{\infty} p \right\}$

⇓ Ersetze y_q^{π} durch $\tilde{y}$.

$= [\circledcirc, \tilde{y}, p, A_w']$ mit $A_w' = \left\{ \sum_{i=1}^{\infty} \left[\sigma_{u_i}\left(\tilde{y}_p^{\omega_i}\right), \omega_i\right] \;\middle|\; \sum_{i=1}^{\infty} u_i \in \Delta_w^{\circledcirc}, \sum_{i=1}^{\infty} \omega_i = \Omega_w(\tilde{y}, p) + \sum_1^{\infty} p \right\}$

⇓ Struktur der Folgezustände des Ausdrucks $\tilde{x} \equiv \underset{p}{\circledcirc}\, \tilde{y} \equiv \underset{p}{\circledcirc}\, y_q^{\pi} \equiv x_q^{\pi}$.

$= \sigma_w(\tilde{x}) = \sigma_w\left(x_q^{\pi}\right)$.

B.3 Korrektheit des optimierten Zustandsmodells

B.3.1 Vorbereitungen

B.3.1.1 Lemma

Wenn zwei Zustände s und $\hat{s}$ äquivalent sind, so sind auch ihre transformierten Zustände $s' = \tau_a(s)$ und $\hat{s}' = \tau_a(\hat{s})$ für jede Aktion $a \in \Sigma$ äquivalent.

Beweis

Für ein Wort $w \in \Sigma^*$ gilt:

$$\psi_w(s') = \psi(\tau_w(s')) = \psi\big(\tau_w(\tau_{\langle a \rangle}(s))\big)$$

⇓ $\tau_v(\tau_u(s)) = \tau_{uv}(s)$ für $u, v \subseteq \Sigma^*$ (Induktion nach der Länge von v).

$$= \psi\big(\tau_{\langle a \rangle w}(s)\big)$$

⇓ Äquivalenz von s und $\hat{s}$.

$$= \psi\big(\tau_{\langle a \rangle w}(\hat{s})\big)$$

⇓ Wie oben.

$$= \psi_w(\hat{s}'),$$

und analog:

$$\varphi_w(s') = \varphi_w(\hat{s}').$$

Daraus folgt wie behauptet die Äquivalenz von s' und $\hat{s}'$.

B.3.1.2 Lemma

Ein optimiertes Zustandsmodell, bestehend aus den Funktionen σ, τ, ψ, φ und ρ, ist korrekt, wenn das zugehörige einfache Modell (bestehend aus den Funktionen σ, τ, ψ und φ) korrekt ist und wenn für jeden Zustand $s \in \Theta$ und jedes Wort $w \in \Sigma^*$ gilt:

$$\hat{\tau}_w(s) \cong \tau_w(s).$$

Beweis

Durch Einsetzen von $s = \sigma(x)$ erhält man für jeden Ausdruck x die Äquivalenz:

$$\hat{\sigma}_w(x) \cong \sigma_w(x),$$

und somit:

$$\hat{\psi}_w(x) = \psi(\hat{\sigma}_w(x)) = \psi(\sigma_w(x)) = \psi_w(x) \quad \text{und} \quad \hat{\varphi}_w(x) = \varphi(\hat{\sigma}_w(x)) = \varphi(\sigma_w(x)) = \varphi_w(x),$$

d. h. die Prädikate ψ und φ der optimierten Zustände stimmen mit denen der einfachen Zustände überein. Daraus folgt die Behauptung.

B.3.1.3 Satz

Ein optimiertes Zustandsmodell, bestehend aus den Funktionen σ, τ, ψ, φ und ρ, ist korrekt, wenn das zugehörige einfache Modell (bestehend aus den Funktionen σ, τ, ψ und φ) korrekt und die Optimierungsfunktion ρ transparent ist (vgl. § 4.5.2.5).

Beweis

Zeige durch vollständige Induktion nach der Länge von w, daß für jeden Zustand $s \in \Theta$ und jedes Wort $w \in \Sigma^*$ die Äquivalenz $\hat{\tau}_w(s) \cong \tau_w(s)$ gilt:

1. Für $w = \langle\rangle$ gilt:

   $$\hat{\tau}_w(s) = s \cong s = \tau_w(s).$$

2. Für $w = \tilde{w}\,\langle a \rangle$ mit $\tilde{w} \in \Sigma^*$ und $a \in \Sigma$ gilt:

   $$\hat{\tau}_w(s) = \hat{\tau}_a\big(\hat{\tau}_{\tilde{w}}(s)\big) = \rho\big(\tau_a\big(\hat{\tau}_{\tilde{w}}(s)\big)\big)$$

   ⇓ Transparenz der Optimierungsfunktion ρ.

   $$\cong \tau_a\big(\hat{\tau}_{\tilde{w}}(s)\big)$$

   ⇓ Nach Induktionsvoraussetzung sind die Zustände $\hat{\tau}_{\tilde{w}}(s)$ und $\tau_{\tilde{w}}(s)$ äquivalent. Aufgrund
   ⇓ des ersten Lemmas (§ B.3.1.1) sind daher auch die transformierten Zustände $\tau_a\big(\hat{\tau}_{\tilde{w}}(s)\big)$
   ⇓ und $\tau_a\big(\tau_{\tilde{w}}(s)\big)$ äquivalent.

   $$\cong \tau_a\big(\tau_{\tilde{w}}(s)\big) = \tau_w(s).$$

Aufgrund des zweiten Lemmas (§ B.3.1.2) folgt hieraus die Behauptung des Satzes.

B.3.2 Transparenz- und Korrektheitstheorem

B.3.2.1 Satz

Die in § 4.5 definierte Optimierungsfunktion ρ ist transparent. Daher ist das dort definierte *optimierte Zustandsmodell* korrekt.

B.3.2.2 Beweis

1. Offensichtlich ist die zusammengesetzte Funktion ρ transparent, wenn die drei Teilfunktionen ρ_1', ρ_2' und ρ_3' transparent sind. Da die identische Funktion trivialerweise transparent ist, genügt es hierfür zu zeigen, daß die partiellen Funktionen ρ_i ($i = 1, 2, 3$) transparent sind, d. h. daß $\rho_i(s) \cong s$ für alle Zustände s gilt, für die $\rho_i(s)$ definiert ist.
2. Da die Folgezustände eines ungültigen Zustands ebenfalls ungültig sind und ein ungültiger Zustand kein Endzustand ist (vgl. § 4.5.1.3), ist ein ungültiger Zustand s offensichtlich äquivalent zum Zustand $\perp$. Daher ist die Funktion ρ_3 transparent.
3. Die Transparenz der Funktionen ρ_1 und ρ_2 wird im folgenden exemplarisch für die sequentielle Komposition gezeigt. Für die übrigen Ausdruckskategorien, für die diese Funktionen definiert sind, verlaufen die Beweise prinzipiell ähnlich.

Beispiel: Sequentielle Komposition

1. Sei $s = [-, z, l, R]$ ein sequentieller Kompositions-Zustand und $\hat{s} = \rho_1(s) = \big[-, z, l, \hat{R}\big]$ mit $\hat{R} = \{\, r \in R \mid \psi(r) \,\}$ der zugehörige optimierte Zustand.
   Aus der Definition der Zustandsübergangsfunktion τ_a folgt durch Induktion nach der Länge eines Wortes $w \in \Sigma^*$:

   $$\tau_w(s) = [-, z, \tau_w(l), R_w] \quad \text{mit} \quad R_w = \{\, \tau_w(r) \mid r \in R \,\} \cup R_w'$$

   und:

   $$\tau_w(\hat{s}) = \big[-, z, \tau_w(l), \hat{R}_w\big] \quad \text{mit} \quad \hat{R}_w = \big\{\, \tau_w(r) \,\big|\, r \in \hat{R} \,\big\} \cup R_w' = \{\, \tau_w(r) \mid r \in R, \psi(r) \,\} \cup R_w',$$

mit einer hier nicht näher interessierenden Menge R'_w, die lediglich von den Folgezuständen von l abhängt und daher für $\tau_w(s)$ und $\tau_w(\hat{s})$ gleich ist.
Da die Folgezustände $\tau_w(r)$ eines ungültigen Zustands r ebenfalls ungültig (und daher auch keine Endzustände) sind, folgt daraus:

$$\psi_w(\hat{s}) = \psi_w(l) \vee \bigvee_{r \in \hat{R}} \psi_w(r) \vee \bigvee_{r \in R'_w} \psi(r) = \psi_w(l) \vee \bigvee_{r \in R} \psi_w(r) \vee \bigvee_{r \in R'_w} \psi(r) = \psi_w(s),$$

und analog:

$$\varphi_w(\hat{s}) = \varphi_w(s),$$

d. h. $\hat{s} \cong s$. Daraus folgt die Transparenz von ρ_1.

2. Sei nun $s = [-, z, l, R]$ mit $R = \{ r \}$ und $\psi(l) = \bot$ sowie $\hat{s} = \rho_2(s) = r$.
Dann gilt für die Folgezustände $\tau_w(s)$ von s:

$$\tau_w(s) = [-, z, \tau_w(l), \{ \tau_w(r) \}] \quad \text{mit} \quad \psi_w(l) = \bot.$$

Daraus folgt:

$$\psi_w(s) = \psi_w(l) \vee \psi_w(r) = \psi_w(r) \quad \text{und} \quad \varphi_w(s) = \varphi_w(r),$$

d. h. $s \cong r = \hat{s}$.

3. Für $s = [-, z, l, R]$ mit $R = \{ r \}$ und $l = \top$ gilt ebenfalls:

$$\tau_w(s) = [-, z, \tau_w(l), \{ \tau_w(r) \}] \quad \text{mit} \quad \psi_w(l) = \bot,$$

allerdings nur für $w \neq \langle\rangle$.
Für das leere Wort $w = \langle\rangle$ ergibt sich jedoch:

$$\psi_w(s) = \psi(s) = \psi(l) \vee \psi(r) = \psi(\top) \vee \psi(r) = \top$$

Da $l = \top$ ein Endzustand ist, enthält die Zustandsmenge R nach Definition den initialen Zustand $\sigma(z)$ des rechten Teilausdrucks z der sequentiellen Komposition $x \equiv y - z$, d. h. es gilt offensichtlich $r = \sigma(z)$.
Da das leere Wort auf jeden Fall ein partielles Wort von z darstellt (vgl. 3.4.4), ist der initiale Zustand $r = \sigma(z)$ gültig, d. h. es gilt $\psi(r) = \top$.

$$= \psi(r) = \psi_w(r)$$

und:

$$\varphi_w(s) = \varphi(s) = \varphi(r) = \varphi_w(r),$$

so daß tatsächlich für alle Worte $w \in \Sigma^*$ gilt:

$$\psi_w(s) = \psi_w(r) \quad \text{und} \quad \varphi_w(s) = \varphi_w(r),$$

d. h. $s \cong r = \hat{s}$. Zusammen mit Punkt 2 folgt daraus die Transparenz von ρ_2.

B.4 Injektive und fokussierte Ausdrücke

Dieser Abschnitt enthält den Beweis des Satzes über injektive und fokussierte Ausdrücke aus § 4.7.2.3.

Vorüberlegungen

Um die Injektivität eines Ausdrucks x zu zeigen, betrachte man eine Aktion $a \in \Sigma$ und zwei Zustände $s_1, s_2 \in \Theta(x)$, die beide die Aktion a akzeptieren, d. h. für die die Folgezustände $s'_1 = \tau_a(s_1)$ und $s'_2 = \tau_a(s_2)$ gültig sind. Dann ist zu zeigen, daß $s_1 = s_2$ gilt.

Da ein ungültiger Zustand keine Aktionen akzeptiert, genügt es hierbei, *gültige* Zustände s_1 und s_2 zu betrachten, d. h. Zustände, die durch die Verarbeitung eines partiellen Worts des Ausdrucks x entstanden sind. Außerdem genügt es, Aktionen $a \in \alpha(x)$ zu betrachten, da nur solche Aktionen von einem Zustand des Ausdrucks x erfolgreich verarbeitet werden können (vgl. § 3.4.4.2).

B.4.1 Atomare Ausdrücke

Für einen atomaren Ausdruck $x \equiv b$ folgt die Behauptung, daß er injektiv und beidseitig fokussiert ist, unmittelbar aus seiner operationalen Definition (vgl. § 4.5.4.1).

B.4.2 Sequentielle Komposition

B.4.2.1 Injektivität

Für eine sequentielle Komposition $x \equiv y - z$ folgt aus den entsprechenden Definitionen für σ, τ und ρ (vgl. § 4.5.4.6):

1. Da der Teilausdruck y terminal fokussiert ist, besitzt er höchstens den Zustand $\top$ als Endzustand, der jedoch bei keinem gewöhnlichen Ausdruck als *initialer* Zustand auftritt.[4] Somit ist der initiale Zustand $\sigma(y)$ von y mit Sicherheit *kein Endzustand*, und daher gilt für den initialen Zustand von x:

   $$\sigma(x) = [-, z, \sigma(y), \emptyset].$$

2. Für einen Zustand $s = [-, z, l, \emptyset]$ gilt $s' = \tau_a(s) = [-, z, l', \emptyset]$, solange der transformierte Zustand $l' = \tau_a(l)$ kein Endzustand ist, und $\hat{s}' = \rho(s') = s'$, sofern l' ein gültiger Zustand ist. Für ein Wort w, das zwar ein *partielles*, aber noch *kein vollständiges* Wort von y darstellt, folgt daher durch vollständige Induktion:

   $$\hat{\sigma}_w(x) = [-, z, \hat{\sigma}_w(y), \emptyset].$$

3. Ist der Zustand $l' = \tau_a(l)$ jedoch ein Endzustand, so ist er aufgrund der terminalen Fokussierung von y gleich $\top$, und es gilt $s' = \tau_a(s) = [-, z, \top, \{\, \sigma(z)\,\}]$ sowie $\hat{s}' = \rho(s') = \sigma(z)$. Daher gilt für ein *vollständiges Wort* w von y:

   $$\hat{\sigma}_w(x) = \sigma(z).$$

4. Wendet man auf diesen Zustand weitere Zustandsübergänge $\hat{\tau}_a$ an, so erhält man offensichtlich Folgezustände $\hat{\sigma}_v(z)$ des Teilausdrucks z. Für ein Wort $w = u\,v$, das ein *vollständiges Wort* u von y *als Präfix* enthält, gilt daher:

   $$\hat{\sigma}_w(x) = \hat{\sigma}_v(z).$$

Zusammenfassend gilt somit für die optimierten Folgezustände $\hat{\sigma}_w(x)$ des Ausdrucks x:

$$\hat{\sigma}_w(x) = \begin{cases} [-, z, \hat{\sigma}_w(y), \emptyset] & \text{für } w \in \Psi(y) \setminus \Phi(y), \qquad \text{(A)} \\ \hat{\sigma}_v(z) & \text{für } w = u\,v \text{ mit } u \in \Phi(y). \qquad \text{(B)} \end{cases}$$

Seien nun s_1 und s_2 zwei derartige Zustände, die beide die Aktion a akzeptieren. Dann können die folgenden drei Fälle unterschieden werden:

1. s_1 und s_2 sind beide von der Gestalt A, d. h. es gilt $s_i = [-, z, l_i, \emptyset]$ und somit $s_i' = \tau_a(s_i) = [-, z, l_i', R_i]$ mit $l_i' = \tau_a(l_i)$ und $R_i = \emptyset$ oder $R_i = \{\, \sigma(z)\,\}$ für $i = 1, 2$.
   Da die Zustände s_1 und s_2 beide die Aktion a akzeptieren, müssen die transformierten Zustände s_i' beide gültig sein, was nur der Fall ist, wenn die Teilzustände l_i' beide gültig sind;[5] dies wiederum

[4] Eine Ausnahme stellt der leere Ausdruck ε dar, der aber nur als Teilausdruck einer Disjunktion auftritt, um eine Option zu simulieren (vgl. § 4.5.4.3).
[5] Beachte: Wenn ein Zustand l_i' ungültig ist, gilt $R_i = \emptyset$, weil l_i' in diesem Fall kein Endzustand ist.

ist nur der Fall, wenn l_1 und l_2 beide die Aktion a akzeptieren. Daraus folgt aufgrund der Injektivität von y die Gleichheit von l_1 und l_2 und somit auch die Gleichheit von s_1 und s_2.

2. s_1 und s_2 sind beide von der Gestalt B, d. h. es gilt $s_i \in \Theta(z)$ für $i = 1, 2$. Da diese Zustände beide die Aktion a akzeptieren, müssen sie aufgrund der Injektivität von z gleich sein.

3. s_1 ist von der Gestalt A (d. h. $s_1 = [-, z, l, \emptyset]$ mit $l \in \Theta(y)$) und s_2 von der Gestalt B (d. h. $s_2 = r \in \Theta(z)$), oder umgekehrt.
   Da s_1 und s_2 beide die Aktion a akzeptieren, müssen − wie oben erläutert − die Zustände $l \in \Theta(y)$ und $r \in \Theta(z)$ diese Aktion ebenfalls akzeptieren. Aufgrund der Voraussetzung $\alpha(y) \cap \alpha(z) = \emptyset$ ist dies jedoch nicht möglich[6], so daß dieser Fall nicht eintreten kann.

Zusammenfassend ergibt sich, daß zwei Zustände $s_1, s_2 \in \hat{\Theta}(x)$, die beide die Aktion a akzeptieren, gleich sind. Daraus folgt die Injektivität des Ausdrucks x.

B.4.2.2 Initiale Fokussierung

Ist der Teilausdruck y initial fokussiert, d. h. gilt $\hat{\sigma}_w(y) \neq \sigma(y)$ für alle $w \neq \langle\rangle$, so gilt offensichtlich:

$$\hat{\sigma}_w(x) = \left\{ \begin{array}{l} [-, z, \hat{\sigma}_w(y), \emptyset] \\ \hat{\sigma}_v(z) \end{array} \right\} \neq [-, z, \sigma(y), \emptyset] = \sigma(x),$$

d. h. auch der Ausdruck x ist dann initial fokussiert.

B.4.2.3 Terminale Fokussierung

Da die Zustände der Gestalt A keine Endzustände sind, besitzt der Ausdruck x offensichtlich dieselben Endzustände wie der Teilausdruck z. Daher folgt aus der terminalen Fokussierung von z auch die terminale Fokussierung von x.

B.4.2.4 Spezialfall

Die Behauptung, daß eine Sequenz $a_1 - \ldots - a_n$ von paarweise verschiedenen Aktionen $a_1, \ldots, a_n$ injektiv und beidseitig fokussiert ist, folgt nun unmittelbar durch vollständige Induktion nach n, wobei die Aussage über atomare Ausdrücke den Induktionsanfang darstellt.

B.4.3 Sequentielle Iteration

Für eine sequentielle Iteration $x \equiv \ominus\, y$ folgt aus der terminalen Fokussierung von y ähnlich wie oben bei der sequentiellen Komposition:

$$\hat{\sigma}_w(x) = \begin{cases} [\ominus, y, \{\, \sigma(y), \top \,\}] & \text{für } w \in \Phi(y)^*, \qquad \text{(A)} \\ [\ominus, y, \{\, \hat{\sigma}_v(y) \,\}] & \text{für } w = uv \text{ mit } u \in \Phi(y)^* \text{ (maximal lang) und } v \neq \langle\rangle. \qquad \text{(B)} \end{cases}$$

Für zwei Zustände $s_1, s_2 \in \hat{\Theta}(x)$, die beide die Aktion a akzeptieren, lassen sich dann ebenfalls drei Fälle unterscheiden:

1. s_1 und s_2 sind beide von der Gestalt A.
   Dann gilt offensichtlich $s_1 = [\ominus, y, \{\, \sigma(y), \top \,\}] = s_2$.

2. s_1 und s_2 sind beide von der Gestalt B.
   Dann folgt mit denselben Argumenten wie in Fall 1 der sequentiellen Komposition aus der Injektivität von y die Gleichheit von s_1 und s_2.

[6] Beachte: Ein Zustand $l \in \Theta(y)$ kann grundsätzlich nur Aktionen $a \in \alpha(y)$ akzeptieren, während ein Zustand $r \in \Theta(z)$ nur Aktionen $a \in \alpha(z)$ akzeptiert.

3. s_1 ist von der Gestalt A und s_2 von der Gestalt B, oder umgekehrt.
   In diesem Fall folgt aus der Injektivität von y die Gleichheit der Teilzustände $\sigma(y)$ von s_1 und $\hat{\sigma}_v(y)$ von s_2, die jedoch im Widerspruch zur initialen Fokussierung von y steht ($\hat{\sigma}_v(y) \neq \sigma(y)$ für $v \neq \langle\rangle$). Daher kann dieser Fall nicht auftreten.

B.4.4 Disjunktion

B.4.4.1 Injektivität

Für den initialen Zustand einer Disjunktion $x \equiv y \bigcirc z$ gilt:

$$\sigma(x) = [\bigcirc, \sigma(y), \sigma(z)].$$

Da die Alphabete der beiden Teilausdrücke y und z disjunkt sind, kann eine Aktion b nur von *einem* der beiden Teilzustände $\sigma(y)$ oder $\sigma(z)$ erfolgreich verarbeitet werden, während der andere beim ersten Zustandsübergang in einen ungültigen Zustand übergeht. Daher gilt für die optimierten Folgezustände des Ausdrucks x offensichtlich:

$$\hat{\sigma}_w(x) = \begin{cases} [\bigcirc, \sigma(y), \sigma(z)] & \text{für } w = \langle\rangle, & \text{(A)} \\ \hat{\sigma}_w(y) & \text{für } w = \langle b, \ldots\rangle \text{ mit } b \in \alpha(y), & \text{(B)} \\ \hat{\sigma}_w(z) & \text{für } w = \langle b, \ldots\rangle \text{ mit } b \in \alpha(z). & \text{(C)} \end{cases}$$

Betrachtet man wieder zwei Zustände $s_1, s_2 \in \hat{\Theta}(x)$, die beide die Aktion a akzeptieren, so können folgende Fälle unterschieden werden:

1. s_1 und s_2 sind beide von der Gestalt A.
   In diesem Fall gilt offensichtlich $s_1 = s_2$.
2. s_1 und s_2 sind beide von der Gestalt B.
   In diesem Fall folgt aus der Injektivität von y unmittelbar die Gleichheit von s_1 und s_2.
   Entsprechendes gilt, wenn s_1 und s_2 beide von der Gestalt C sind.
3. s_1 ist von der Gestalt A und s_2 von der Gestalt B, oder umgekehrt.
   Da die Aktion a von $s_2 = \hat{\sigma}_w(y)$ akzeptiert wird, muß sie zum Alphabet von y gehören. Da die Alphabete von y und z disjunkt sind, wird sie folglich vom Teilzustand $\sigma(z)$ des Zustands s_1 *nicht* akzeptiert und muß daher vom Teilzustand $\sigma(y)$ akzeptiert werden. Aufgrund der Injektivität von y müßte daher $\sigma(y) = \hat{\sigma}_w(y)$ für ein Wort $w \neq \langle\rangle$ gelten, was im Widerspruch zur initialen Fokussierung von y steht. Daher kann dieser Fall nicht auftreten.
   Entsprechendes gilt, wenn s_2 von der Gestalt C ist.
4. s_1 ist von der Gestalt B und s_2 von der Gestalt C, oder umgekehrt.
   Dieser Fall kann aufgrund der Voraussetzung $\alpha(y) \cap \alpha(z) = \emptyset$ nicht auftreten.

Zusammengefaßt folgt hieraus die Injektivität des Ausdrucks x.

B.4.4.2 Initiale Fokussierung

Die initiale Fokussierung von x folgt unmittelbar aus der obigen Formel für $\hat{\sigma}_w(x)$.

B.4.4.3 Terminale Fokussierung

Wenn die Teilausdrücke y und z beide terminal fokussiert sind, sind ihre initialen Zustände $\sigma(y)$ und $\sigma(z)$ *keine* Endzustände (vgl. die entsprechende Argumentation im Kontext der sequentiellen Komposition in § B.4.2.1). Somit ist auch der initiale Zustand $\sigma(x) = [\bigcirc, \sigma(y), \sigma(z)]$ kein Endzustand. Folglich sind alle Endzustände des Ausdrucks x von der Gestalt B oder C und somit, aufgrund der terminalen Fokussierung von y und z, gleich $\top$.

B.4.5 Option

Die Behauptung für die Option $x \equiv \circlearrowright y$ folgt aus der Beziehung $\circlearrowright y = y \bigcirc \varepsilon$ (vgl. § 4.5.4.3) und der entsprechenden Aussage für Disjunktionen, unter Berücksichtigung der Tatsache, daß der leere Ausdruck ε offensichtlich injektiv, initial fokussiert und disjunkt zu jedem Ausdruck y ist.

Man beachte in diesem Zusammenhang jedoch, daß aus der terminalen Fokussierung des Teilausdrucks y *nicht* etwa die terminale Fokussierung der Option $x \equiv \circlearrowright y$ folgt, obwohl der leere Ausdruck ε ebenfalls terminal fokussiert ist. Dieser scheinbare Widerspruch liegt darin begründet, daß die entsprechende Aussage für Disjunktionen nur für *gewöhnliche* Teilausdrücke $y, z \neq \varepsilon$ zutrifft, deren initialer Zustand *verschieden* vom Zustand $\top$ ist (vgl. § B.4.4.3).

B.4.6 Disjunktions-Quantorausdrücke

B.4.6.1 Injektivität und initiale Fokussierung

Für den initialen Zustand eines Disjunktions-Quantorausdrucks $x \equiv \bigcirc_p y$ gilt:

$$\sigma(x) = [\bigcirc, y, p, \sigma(y), \varnothing].$$

Aufgrund der vollständigen und homogenen Quantifizierung, die gemäß § 4.7.2.2 impliziert, daß die Zweige y_p^ω paarweise disjunkte Alphabete besitzen, folgt ähnlich wie bei der binären Disjunktion in § B.4.4.1, daß eine Aktion b von *höchstens einem* konkretisierten Zweig y_p^ω bzw. Zustand $(\sigma(y))_p^\omega$ erfolgreich verarbeitet werden kann, während der Zustand $\sigma(y)$ des abstrakten Zweigs y beim ersten Zustandsübergang in einen ungültigen Zustand übergeht. Beachtet man zusätzlich die Optimierungsfunktion ρ_1, so gilt daher für Worte $w = \langle b, \ldots \rangle$:

$$\hat{\sigma}_w(x) = \left[\bigcirc, y, p, \bot, \left\{ \left[\hat{\sigma}_w\left(y_p^\omega\right), \omega\right] \right\}\right],$$

wobei ω den einzigen relevanten Parameterwert der Aktion b darstellt (vgl. § 4.7.2.2). Daraus folgt sowohl die Injektivität als auch die initiale Fokussierung des Ausdrucks x ähnlich wie oben bei der binären Disjunktion.

B.4.6.2 Terminale Fokussierung

Ist der Rumpf y terminal fokussiert, so besitzt jeder Zweig y_p^ω höchstens den Zustand $\top$ als Endzustand. Aufgrund der Optimierung

$$\rho_2\left([\bigcirc, y, p, \bot, \{ [\top, \omega] \}]\right) = \top$$

und der Tatsache, daß der initiale Zustand $\sigma(x) = [\bigcirc, y, p, \sigma(y), \varnothing]$ *kein* Endzustand ist, besitzt der Ausdruck x daher höchstens den Zustand $\top$ als Endzustand und ist somit terminal fokussiert.

Anhang C

Ein syntaxgesteuerter Editor für Interaktionsgraphen

C.1 Einleitung

Um die praktische Erstellung von Interaktionsgraphen zu erleichtern, wurde im Rahmen dieser Arbeit ein *syntaxgesteuerter Editor* entwickelt, der sich durch *einfache Bedienbarkeit*, *automatische Formatierung* und *implizite Konsistenzsicherung* der erstellten Graphen auszeichnet. Wesentliche Designentscheidungen bei der Entwicklung des Editors waren:

1. Der Graphautor, d. h. der Benutzer des Editors, soll lediglich die *Struktur* der Graphen erstellen, während ihr *Layout*, d. h. ihre konkrete graphische Darstellung, automatisch vom Programm erzeugt wird. Gegenüber sonst üblichen „Freistil-Editoren“, mit denen graphische Objekte beliebig plaziert und „kreuz und quer“ miteinander verbunden werden können, besitzt dieser Ansatz den Vorteil, daß die erstellten Graphen stets ein *homogenes* und ästhetisch ansprechendes *Erscheinungsbild* besitzen. So sind Verzweigungen beispielsweise immer vollkommen symmetrisch aufgebaut, und die Kanten von Wiederholungen besitzen stets dieselbe Neigung. Außerdem kann sich der Autor eines Graphen ganz auf dessen Inhalt konzentrieren, weil er nicht permanent durch die Gestaltung seiner äußeren Form abgelenkt wird, die ohnehin durch jede inhaltliche Änderung wieder mehr oder weniger zerstört wird. Schließlich kann durch diese strikte Strukturorientierung auf einfache Art und Weise gewährleistet werden, daß alle erstellten Graphen *syntaktisch korrekt* sind.

2. Um neben der rein syntaktischen Korrektheit auch sicherstellen zu können, daß Graphen keine undefinierten Aktionen, Aktivitäten, Abkürzungen oder Schablonen enthalten, können diese Grundelemente nicht frei definiert, sondern nur aus einer vorgegebenen Palette *ausgewählt* werden. Neben den *Namen* der verfügbaren Elemente enthält eine solche Palette − die beispielsweise von einem Systemverwalter erstellt werden kann − auch die Information, wieviele *Parameter* eine Aktion, Aktivität oder Abkürzung besitzt und wieviele *Zweige* eine Ausprägung einer Schablone mindestens oder höchstens enthalten darf. Gegenüber gebräuchlichen Workflow-Editoren, mit denen Aktivitäten während der Erstellung eines Workflows en passant definiert und nach Belieben benannt werden können, besitzt dieser Ansatz den Vorteil, daß Aktivitäten in allen Graphen *einheitliche Bezeichnungen* besitzen und die Verwendung fehlerhafter Namen a priori ausgeschlossen ist (vgl. auch § 5.4.6). Ebenso ist sichergestellt, daß alle Aktivitäten eine *korrekte Anzahl von Parametern* besitzen.

Abgesehen von den bereits genannten Vorteilen, tragen diese Designentscheidungen auch wesentlich dazu bei, daß die Benutzerschnittstelle des Editors einfach zu bedienen und auch relativ leicht zu implementieren ist. Da graphische Elemente beispielsweise nicht frei plaziert werden dürfen, entfällt die Notwendigkeit entsprechender Bedienelemente, was sowohl die Handhabung als auch die Implementierung des Programms vereinfacht.

C.2 Benutzerschnittstelle

Im folgenden sollen die Gestaltung der graphischen Oberfläche sowie die wesentlichen Bedienprinzipien des Editors kurz erläutert werden, indem der Beispielgraph in Abb. C.1 schrittweise erstellt wird. Allerdings stellen diese Ausführungen kein vollständiges Benutzerhandbuch dar, was zur Folge hat, daß an der einen oder anderen Stelle möglicherweise Detailfragen unbeantwortet bleiben.

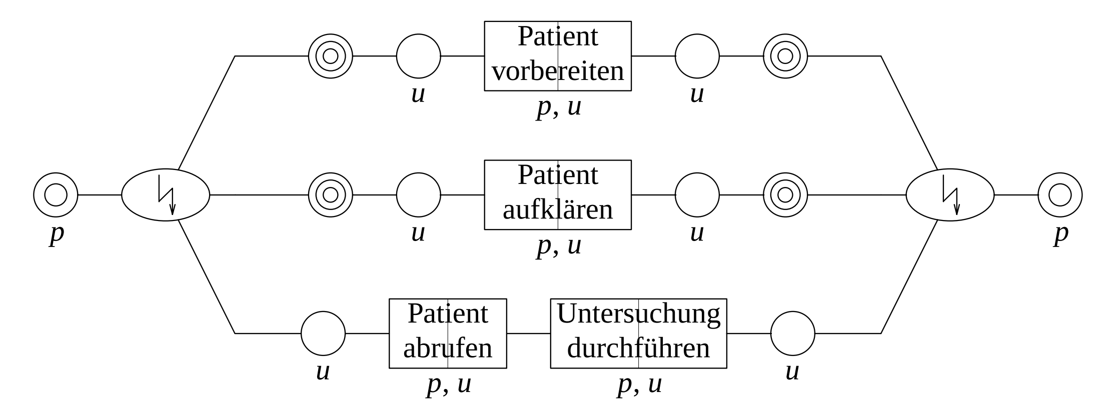


Abbildung C.1: Beispielgraph (vgl. Abb. 2.62, § 2.7.1.2)

C.2.1 Start des Programms

Abbildung C.2 zeigt die Oberfläche des Programms unmittelbar nach dem Start:

- Die erste „Zeile“ (d. h. das oberste Rechteck), auf die in § C.2.7 genauer eingegangen wird, stellt den Pfad des *aktuellen Verzeichnisses* (im Beispiel `/home/dbis/heinlein/graphen`) als *Pseudograph* dar.
- Die nächsten beiden Zeilen enthalten die *Standardoperatoren* von Interaktionsgraphen als leere *Mustergraphen*:

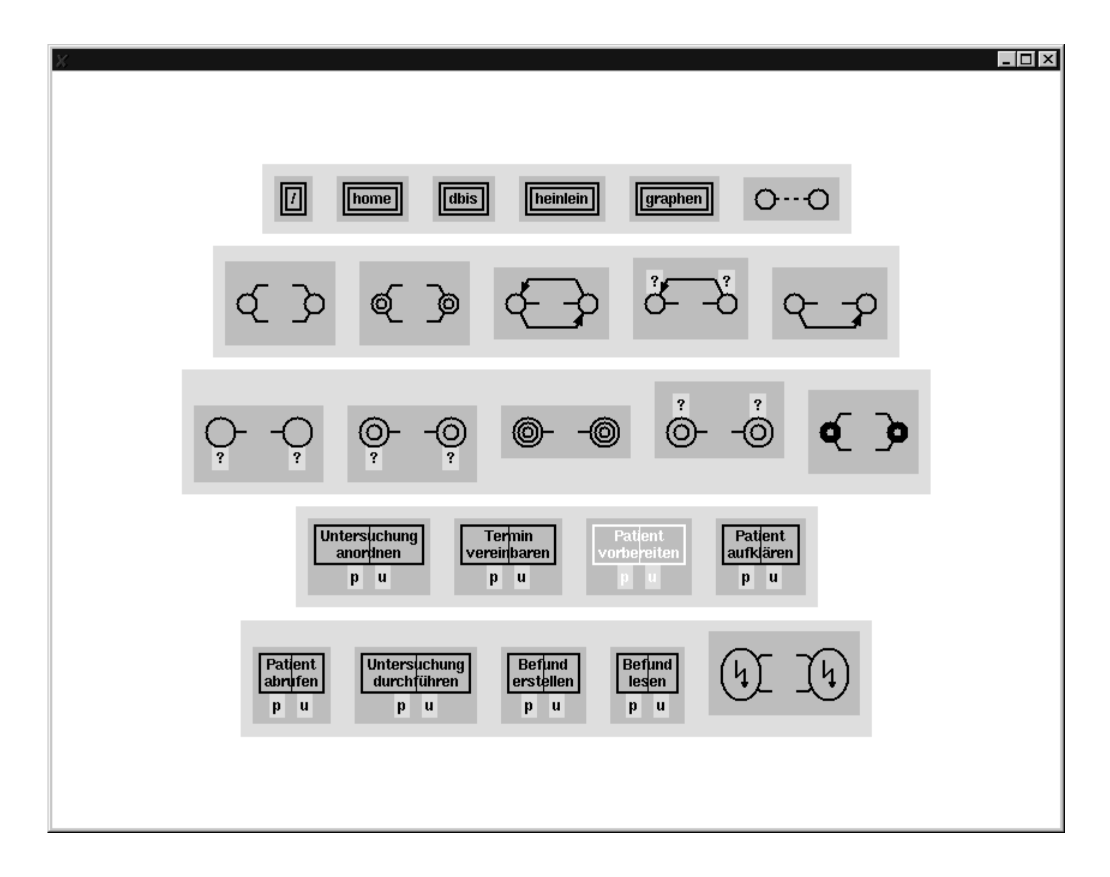


Abbildung C.2: Oberfläche nach dem Start des Programms

  - ◦ Entweder-oder-Verzweigung, Sowohl-als-auch-Verzweigung, Wiederholung, Mehrfach-Ausführung, Eventuell-Verzweigung;
  - ◦ Für-ein-Verzweigung, Für-alle-Verzweigung, Beliebig-oft-Verzweigung, Mehrfach-Verzweigung, Kopplung.
- Die letzten beiden Zeilen enthalten als *vordefinierte Grundelemente* die in Tab. 5.31 (§ 5.4.2) genannten Aktivitäten sowie die Schablone für den wechselseitigen Ausschluß (vgl. § 2.3.3.1).

Sowohl die Liste der Standardoperatoren als auch die Palette der vordefinierten Grundelemente kann über Kommandozeilenargumente oder Konfigurationsdateien angepaßt werden.

Auf einem Farbbildschirm erscheinen die Graphen normalerweise als *schwarze Linien und Texte* auf *blauem Hintergrund*, der für Sequenzen, Parameter und Faktoren *hell* und für Sequenzelemente, d. h. Verzweigungen und Aktivitäten, *dunkel* gefärbt ist.[1] Auf diese Weise läßt sich die hierarchische Struktur eines Graphen optisch sehr schnell erfassen.

C.2.2 Markieren und Kopieren von Teilgraphen

Wird die Maus im Fenster des Editors bewegt, so wird immer derjenige Teilgraph *invertiert*, d. h. mit *weißen* Linien und Texten dargestellt, der gerade den Mauszeiger enthält. In Abb. C.2 ist dies z. B. die Aktivität Patient vorbereiten. Wird nun mit der *linken* Maustaste geklickt, so wird dieser Teilgraph *markiert*, was optisch durch einen Wechsel der Hintergrundfarbe in ein entsprechendes *Rot* angezeigt wird.

Bewegt man den Mauszeiger nun in den unteren, leeren Bereich des Fensters und drückt dort die *mittlere* Maustaste, so wird der zuvor markierte Teilgraph dorthin *kopiert*. Außerdem geht die Markierung zu dem neu eingefügten Teilgraphen über. Abbildung C.3 zeigt den resultierenden Fensterinhalt der Schwarz-weiß-Version, in der rote Hintergrundbereiche dunkler als blaue dargestellt werden. Die Abbildung zeigt außerdem, daß die kopierte Aktivität Patient vorbereiten (dunkelrot) automatisch mit einer einelementigen Sequenz (hellblau) umgeben wurde.

Will man die markierte Aktivität nun in eine der Standard-Verzweigungen einbetten, so positioniert man den Mauszeiger an die entsprechende Stelle − beispielsweise in die Für-ein-Verzweigung in der dritten Zeile links − und drückt dort die mittlere Maustaste. Auf diese Weise wird der markierte Teilgraph *quasi* dorthin kopiert. Da es sich bei der Verzweigung, die das Ziel dieser Kopieroperation darstellt, jedoch um einen vordefinierten *Mustergraphen* handelt, der nicht verändert werden darf, wird die Operation stattdessen auf einer neu erzeugten *Kopie* des Verzweigungsgraphen ausgeführt, die anschließend an die Stelle des markierten Teilgraphen tritt (vgl. Abb. C.4).

Um der so erzeugten Für-ein-Verzweigung den Parameter u zuzuweisen, kann dieser ebenfalls durch Markieren und Kopieren von der Aktivität Patient vorbereiten übernommen werden, wobei es gleichgültig ist, ob man als Ziel der Kopieroperation das Parameterfeld des linken oder des rechten ○-Operatorsymbols wählt. Ähnlich wie oben, kann nun die gesamte Für-ein-Verzweigung in eine Beliebig-oft-Verzweigung und diese wiederum in einen wechselseitigen Ausschluß eingebettet werden (vgl. Abb. C.5).

C.2.3 Weitere nützliche Operationen

Wenn der erstellte Graph zusammen mit den vordefinierten Mustergraphen größer als das Fenster des Editors wird, kann die Zeichenfläche, auf der sich die Graphen befinden, wie ein Blatt Papier innerhalb des Fensters *verschoben* werden. Anders als mit den weitverbreiteten Rollbalken (engl. scrollbars), deren Benutzung insbesondere im zweidimensionalen Raum äußerst unbefriedigend ist, kann die Zeichenfläche einfach durch Drücken der linken Maustaste gegriffen und bei gedrückter Mausta-

[1] Der Begriff *Aktivität* wird im folgenden als Stellvertreter für Aktionen, Aktivitäten und Abkürzungen verwendet, die alle durch unterschiedliche Arten von *Rechtecken* dargestellt werden.
Alle Farben des Programms können ebenfalls über Kommandozeilenargumente oder Konfigurationsdateien verändert werden. Auf diese Weise kann man problemlos eine Schwarz-weiß-Version starten, mit der die hier gezeigten Bildschirmausdrucke erzeugt wurden.

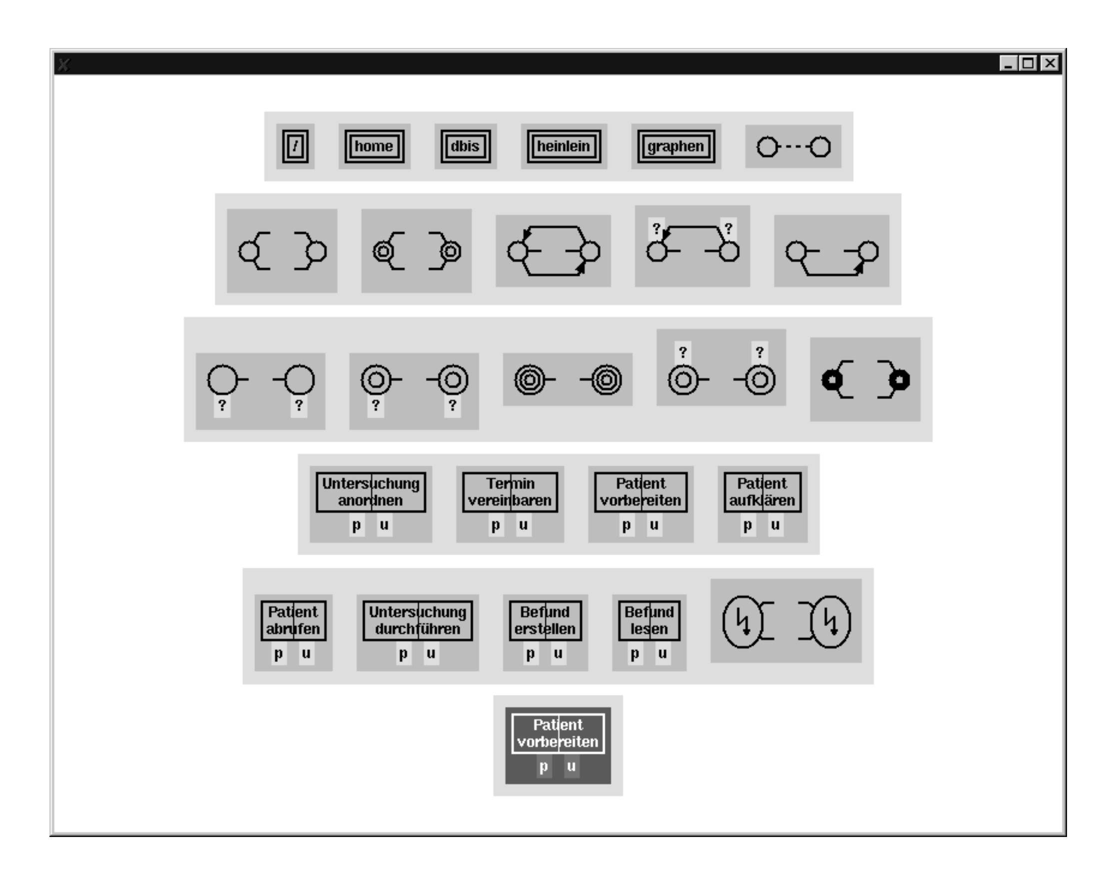


Abbildung C.3: Markieren und Kopieren der Aktivität Patient vorbereiten

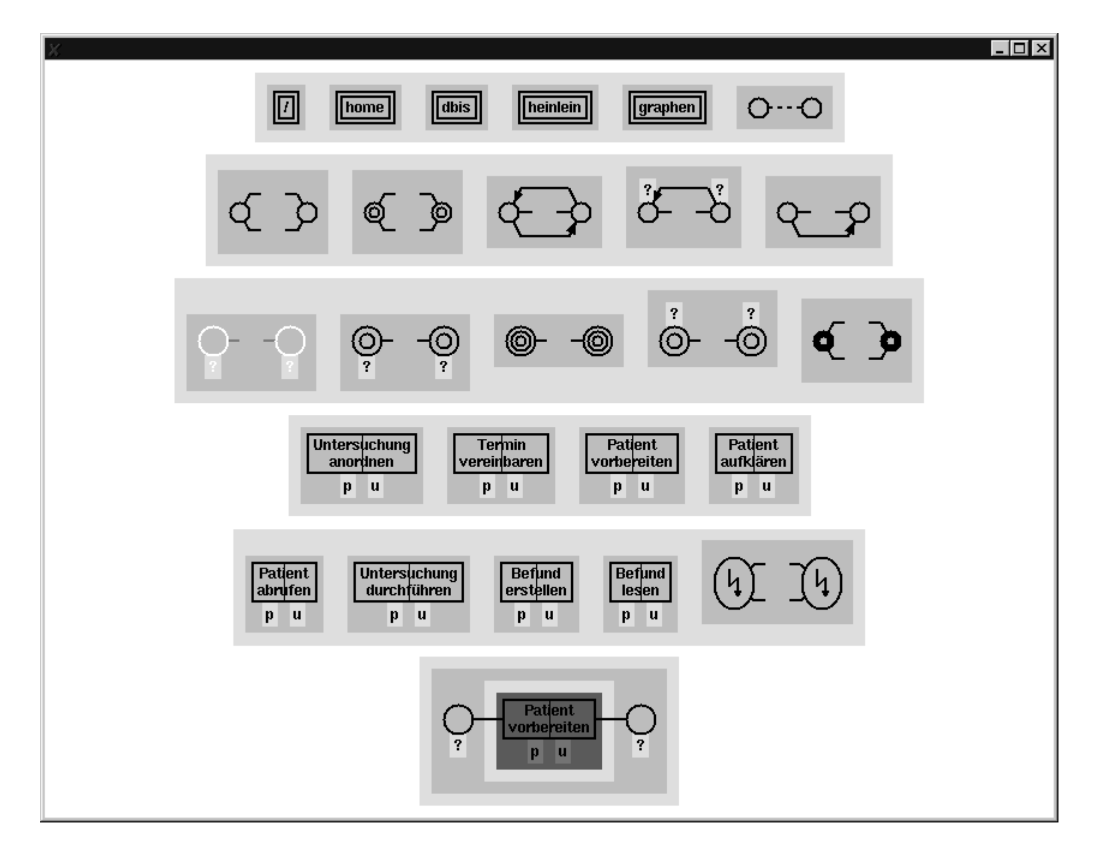


Abbildung C.4: Einbetten der Aktivität Patient vorbereiten in eine Für-ein-Verzweigung

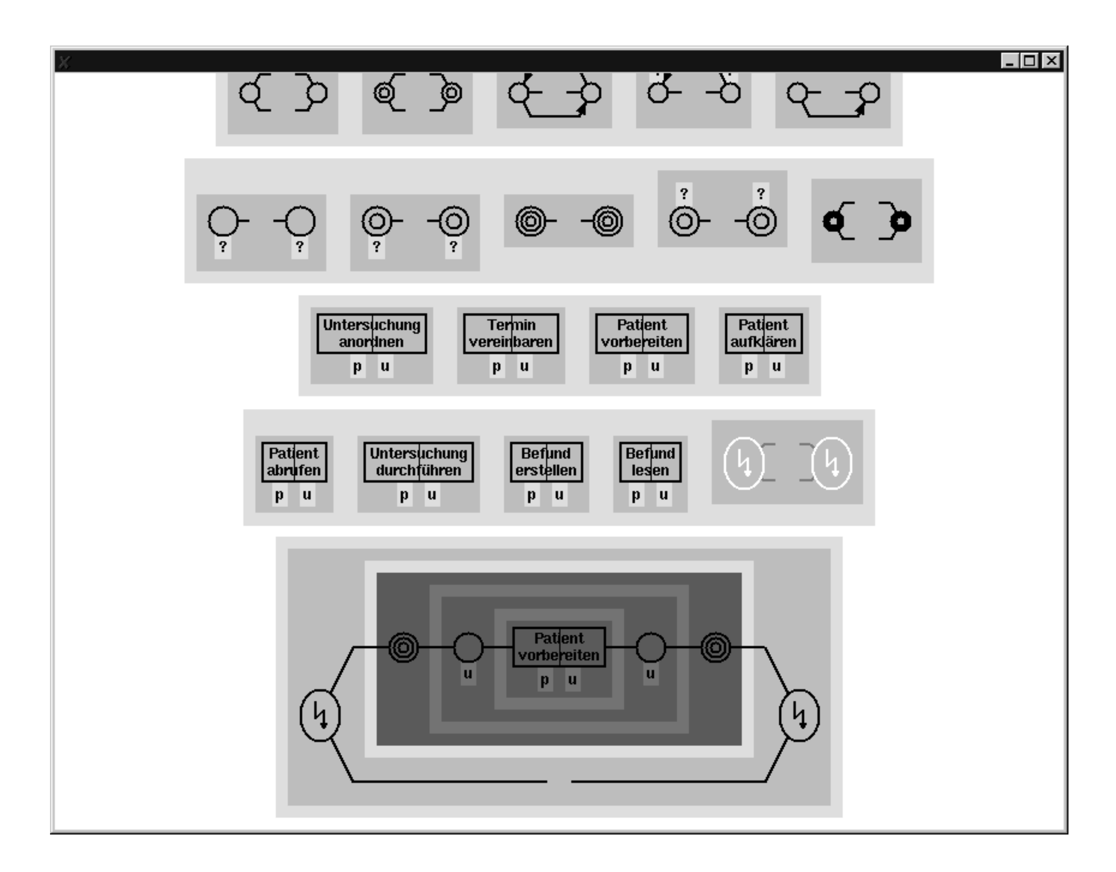


Abbildung C.5: Beliebig-oft-Verzweigung und wechselseitiger Ausschluß

ste beliebig horizontal und vertikal bewegt werden. In Abb. C.5 wurde sie beispielsweise ein Stück nach oben geschoben, damit der bis jetzt konstruierte Graph vollständig sichtbar ist.

Alternativ kann man durch Klicken mit der *rechten* Maustaste einzelne Teilgraphen *zu-* und später wieder *aufklappen*. Beispielsweise ist die Entweder-oder-Verzweigung in Abb. C.4 ganz oben rechts momentan zugeklappt, was durch eine *gestrichelte Linie* zwischen ihren beiden ○-Knoten visualisiert wird.

Eine dritte Möglichkeit, die Größe eines Graphen zu beeinflussen, besteht darin, durch horizontales Bewegen der Maus bei *gedrückter* rechter Maustaste (engl. drag) die *Schriftgröße* und damit indirekt die Abmessungen sämtlicher Graphen schrittweise zu verändern.

Um irrtümlich vorgenommene Änderungsoperationen leicht und schnell *rückgängig machen* zu können, kann man sich durch horizontales Bewegen der Maus bei gedrückter *mittlerer* Maustaste quasi auf der Zeitachse hin- und herbewegen und so zu jeder früheren Version des erstellten Graphen zurückkehren.

Klickt man mit der mittleren Maustaste innerhalb des gerade markierten Teilgraphen, so wird dieser nicht wie sonst kopiert (da Quelle und Ziel der Kopieroperation identisch wären), sondern *entfernt*. Da versehentlich durchgeführte Operationen problemlos rückgängig gemacht werden können, wird eine solche Löschoperation ohne Nachfrage und Bestätigung durchgeführt. Außerdem wird der entfernte Teilgraph intern so lange aufbewahrt, bis eine neue Markieroperation ausgeführt wird, d. h. er kann durch eine anschließende Kopieroperation wieder zum Vorschein gebracht werden. Auf diese Weise kann ein Teilgraph einfach durch Markieren, Entfernen und anschließendes Kopieren an eine andere Stelle *versetzt* werden.

C.2.4 Kopieren größerer Teilgraphen

Der mittlere Zweig des wechselseitigen Ausschlusses in Abb. C.1 kann prinzipiell nach demselben Schema erstellt werden wie der obere. Einfacher und schneller ist es jedoch, den oberen Zweig durch Markieren und Kopieren komplett zu verdoppeln, anschließend die Aktivität Patient vorbereiten zu entfernen und stattdessen eine Kopie von Patient aufklären einzusetzen (vgl. Abb. C.6).

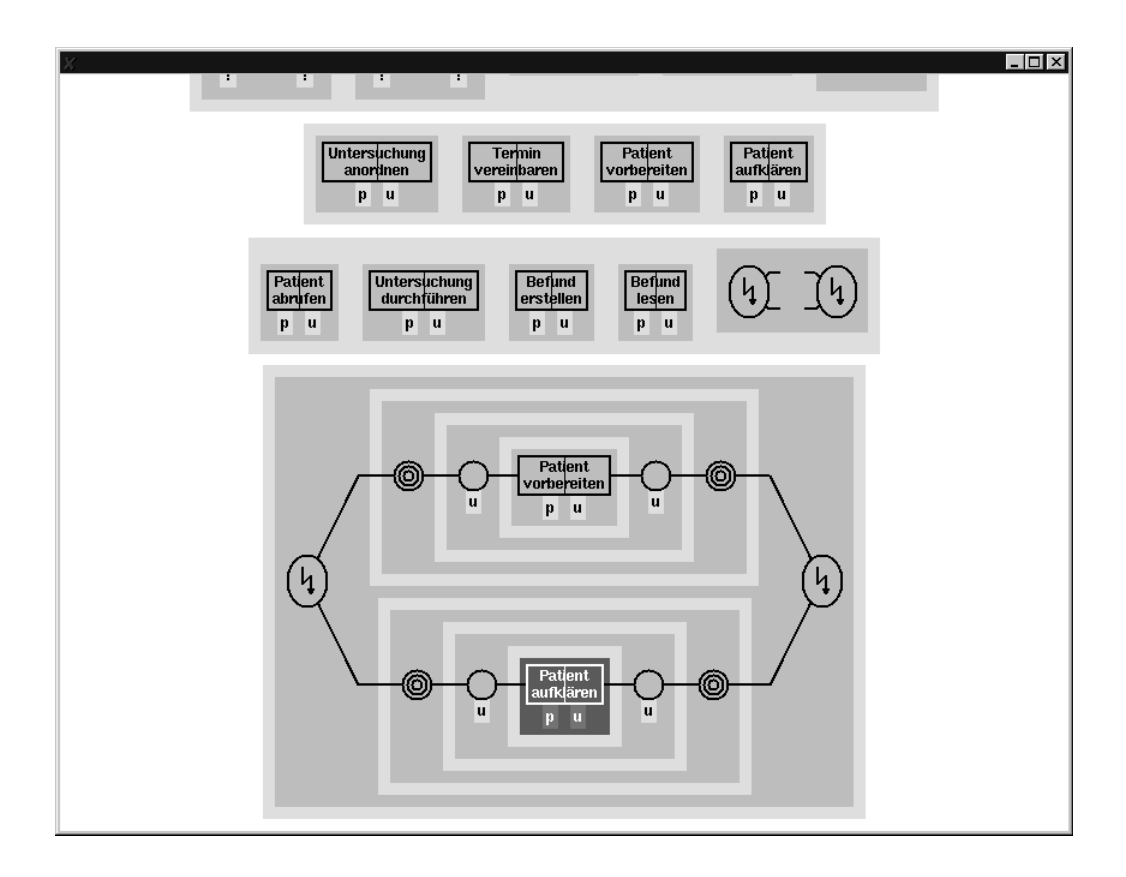


Abbildung C.6: Zweiter Zweig des wechselseitigen Ausschlusses

C.2.5 Konstruktion von Verzweigungen mit mehr als zwei Zweigen

Ebenso wie die Standardverzweigungen von Interaktionsgraphen, kann auch ein wechselseitiger Ausschluß beliebig viele Zweige besitzen (vgl. § 2.3.3.2), obwohl der zugehörige Mustergraph nur Platzhalter für die mindestens benötigten zwei Zweige aufweist.

Um konkret dem wechselseitigen Ausschluß in Abb. C.6 einen dritten Zweig gemäß Abb. C.1 hinzuzufügen, markiert man z. B. das Muster der Für-ein-Verzweigung in der dritten Zeile links und bewegt den Mauszeiger in die bis jetzt konstruierte „Blitzverzweigung“. Klickt man nun mit der mittleren Maustaste, so wird der markierte Teilgraph dem wechselseitigen Ausschluß als oberster (1), mittlerer (2) oder unterster Zweig (3) hinzugefügt, je nachdem ob sich der Mauszeiger oberhalb (1), zwischen (2) oder unterhalb (3) der beiden vorhandenen Zweige befindet. Als optische Orientierungshilfe „leuchtet“ hierbei abhängig von der Mausposition entweder der oberste Zweig (1), beide Zweige (2) oder der untere Zweig (3) rot auf. Abbildung C.7 zeigt den resultierenden Graphen, wenn man sich für Variante 3 entscheidet.

Anmerkung: Bewegt man jetzt den Mauszeiger innerhalb des wechselseitigen Ausschlusses auf und ab, so leuchtet entweder der obere der drei Zweige (1), die oberen beiden Zweige (2), die unteren bei-

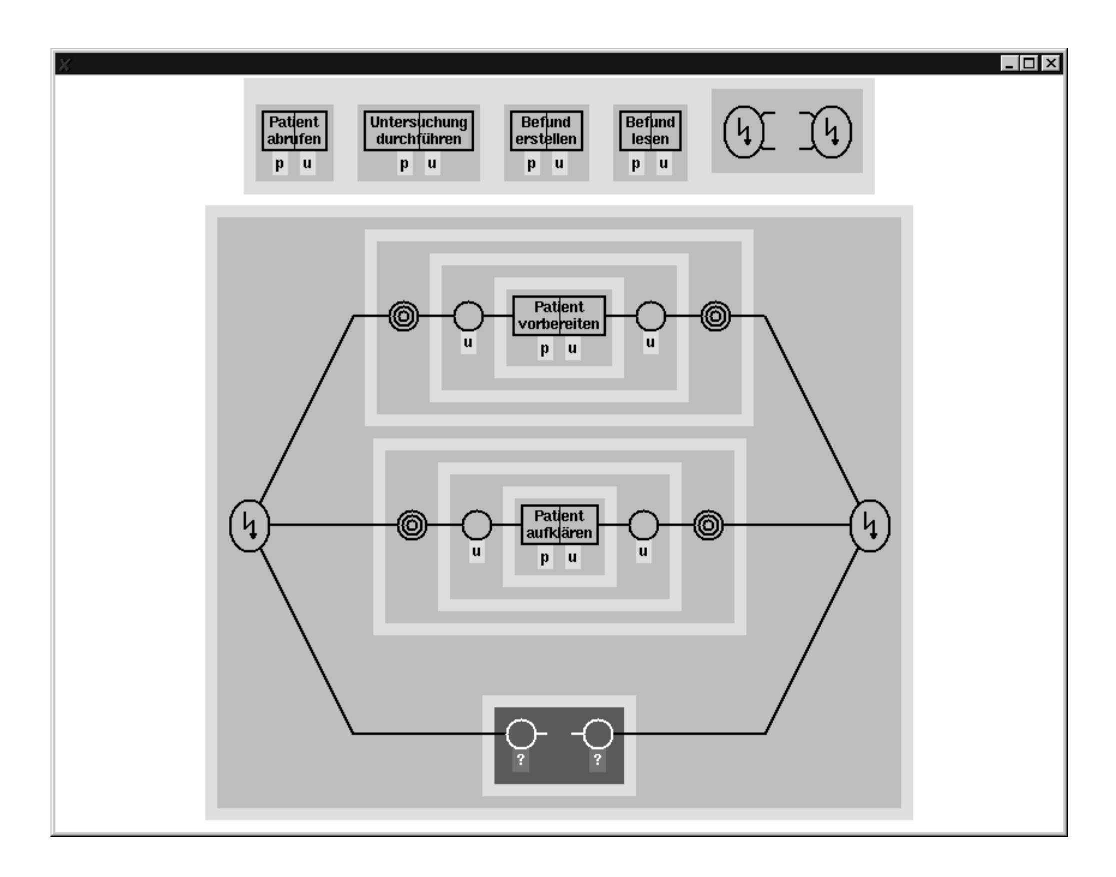


Abbildung C.7: Wechselseitiger Ausschluß mit drei Zweigen

den Zweige (3) oder der unterste Zweig (4) rot auf. Dementsprechend würde eine weitere Kopieroperation einen neuen Zweig ganz oben (1), zwischen den oberen beiden Zweigen (2), zwischen den unteren beiden Zweigen (3) oder ganz unten (4) einfügen usw.

C.2.6 Konstruktion von Sequenzen

Fügt man nun die Aktivität Patient abrufen in die noch leere Für-ein-Verzweigung ein, so erhält man den Graphen in Abb. C.8. Markiert man anschließend die Muster-Aktivität Untersuchung durchführen und plaziert den Mauszeiger an den rechten Rand der Sequenz, die die gerade eingefügte Aktivität Patient abrufen umgibt, so leuchtet dort die horizontale Verbindungslinie zum rechten ○-Operator rot auf. Dadurch wird angezeigt, daß eine anschließende Kopieroperation den markierten Teilgraphen an dieser Stelle als Sequenzelement einfügen wird. Abbildung C.9 zeigt das entsprechende Resultat.

Anmerkung: Bewegt man jetzt den Mauszeiger innerhalb der Sequenz hin und her, so leuchtet entweder die horizontale Verbindungslinie zwischen dem linken ○-Operator und der Aktivität Patient abrufen (1), die Linie zwischen den beiden Aktivitäten (2) oder die Linie zwischen der Aktivität Untersuchung durchführen und dem rechten ○-Operator (3) rot auf (sofern sich der Mauszeiger nicht innerhalb einer dieser Aktivitäten befindet). Dementsprechend würde eine weitere Kopieroperation ein neues Sequenzelement vor der Aktivität Patient abrufen (1), zwischen den beiden Aktivitäten (2) oder nach der Aktivität Untersuchung durchführen (3) einfügen.

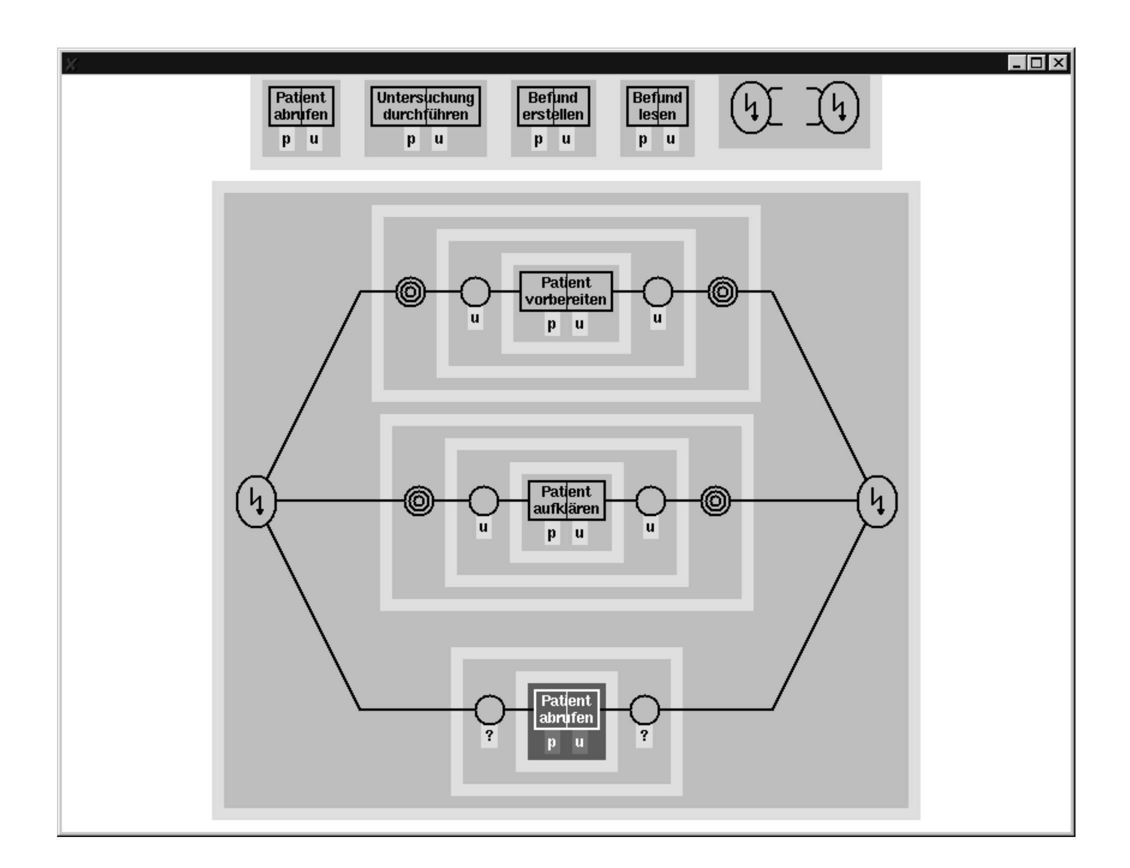


Abbildung C.8: Konstruktion einer Sequenz (1)

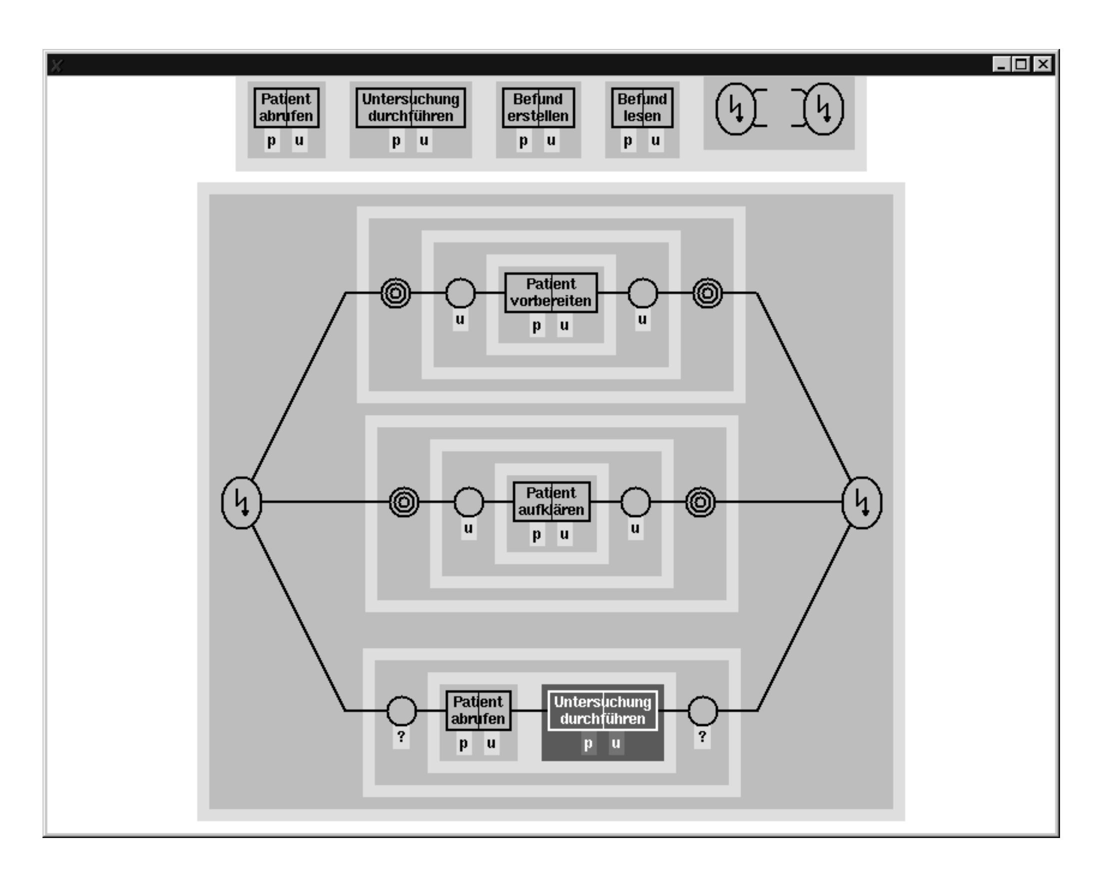


Abbildung C.9: Konstruktion einer Sequenz (2)

C.2.7 Schnittstelle zum Dateisystem

Um den bis jetzt konstruierten Beispielgraphen zu vervollständigen, muß die unterste Für-ein-Verzweigung noch einen Parameter u erhalten und der gesamte wechselseitige Ausschluß in eine Für-alle-Verzweigung mit dem Parameter p eingebettet werden (vgl. Abb. C.10).

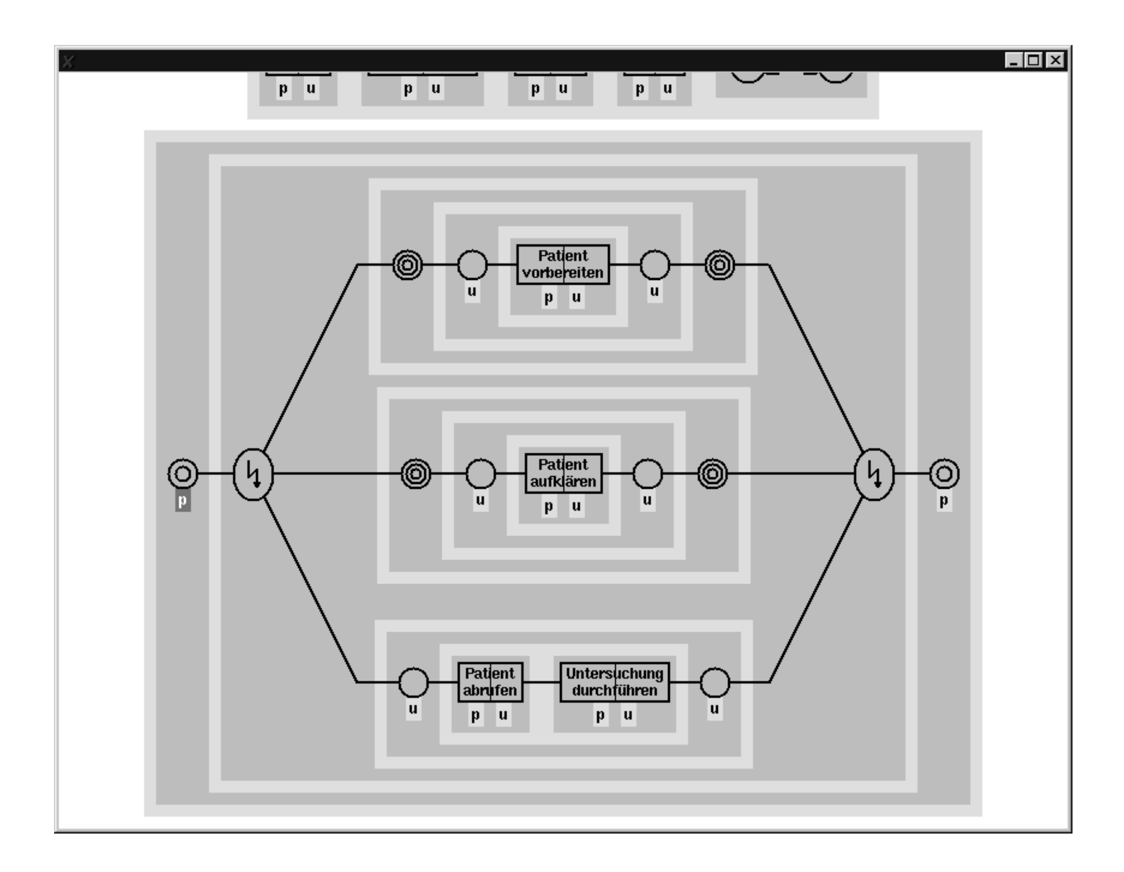


Abbildung C.10: Vollständiger Beispielgraph

Um den so erstellten Graphen in einer Datei abzuspeichern, wird er zunächst wie gewohnt markiert. Anschließend bewegt man den Mauszeiger in die oberste Zeile des Fensters und klappt dort die bis jetzt geschlossene Entweder-oder-Verzweigung auf, die alle Dateien des aktuellen Verzeichnisses als Pseudoaktionen (einfacher Rahmen) und eventuell vorhandene Unterverzeichnisse als Pseudoabkürzungen (Doppelrahmen) enthält (vgl. Abb. C.11). Plaziert man den Mauszeiger in eine „Datei-Aktion", so leuchtet diese rot auf und signalisiert damit, daß sie als Ziel einer anschließenden Kopieroperation dienen kann. Drückt man nun die rechte Maustaste, so wird der markierte Graph in einen äquivalenten Interaktionsausdruck transformiert und so in die Datei geschrieben.

Umgekehrt kann man durch Markieren einer Datei-Aktion und anschließendes Kopieren auf die Zeichenfläche (oder auch in einen bestehenden Graphen) einen Ausdruck aus dieser Datei einlesen und verwenden, nachdem er mit Hilfe eines eingebauten Parsers in einen äquivalenten Graphen verwandelt wurde.

Durch Klicken auf ein Verzeichnis kann man dieses zum aktuellen Verzeichnis machen, was zur Folge hat, daß der Datei-Verzeichnis-Graph entsprechend aktualisiert wird. Um eine neue Datei zu erzeugen, plaziert man den Mauszeiger in die oben erwähnte Entweder-oder-Verzweigung und drückt die Return-Taste, um ein Dialogfenster zur Eingabe eines Dateinamens zu öffnen. Nach Beendigung des Dialogs wird eine entsprechend benannte Datei im aktuellen Verzeichnis erzeugt und der Datei-Verzeichnis-Graph aktualisiert.

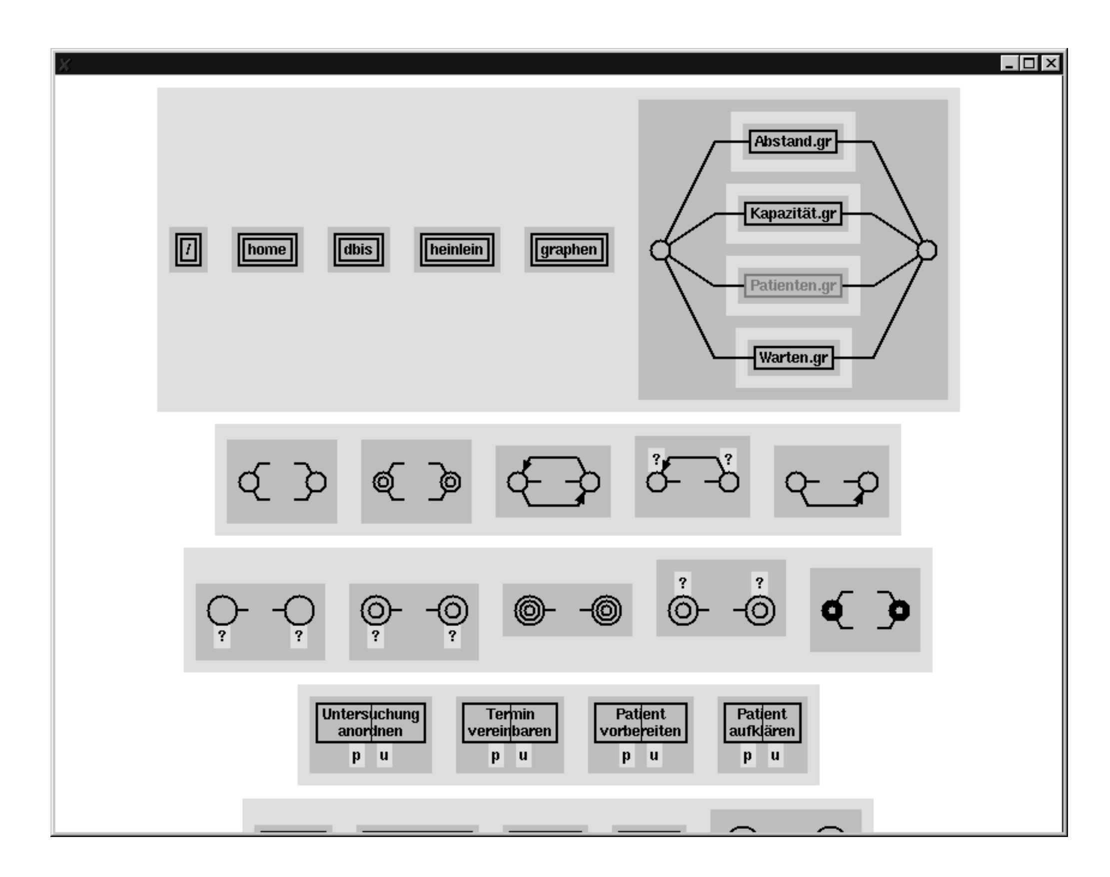


Abbildung C.11: Schnittstelle zum Dateisystem

Anmerkung: Neben dem Erzeugen neuer Dateien gibt es eine zweite, bisher nicht erwähnte Situation, in der die Tastatur als Eingabemedium verwendet werden kann: Drückt man die Return-Taste, während sich der Mauszeiger auf einem Parameter einer Aktivität oder eines Operators befindet, so erhält man ein Dialogfenster zur Eingabe eines neuen Parameterwerts, der den alten Wert des Parameters überschreibt. Dies ist immer dann erforderlich, wenn der Wert eines Parameters nicht wie in § C.2.2 beschrieben durch Markieren oder Kopieren von einer anderen Aktivität übernommen werden kann. Insbesondere kann man auf diese Weise den numerischen Faktor einer Mehrfach-Ausführung oder -Verzweigung eingeben.

C.2.8 Zusammenfassung der Maus-Interaktionen

Tabelle C.12 zeigt die verschiedenen Maus-Interaktionen noch einmal im Zusammenhang. Wie in vielen anderen Programmen auch, kann man mit der *linken* Maustaste entweder Objekte (in diesem Fall Graphen oder Teilgraphen) markieren oder aber „window scrolling“ erreichen. Sämtliche Änderungsoperationen werden mit der *mittleren* Maustaste durchgeführt: durch Klicken wird ein Graph entweder kopiert oder entfernt, und durch Ziehen der Maus kann man sich auf der Zeitachse bewegen, d. h. Änderungen rückgängig („history scrolling“). Mit der *rechten* Maustaste schließlich läßt sich die Grö-

Maustaste	*Klicken*	*Ziehen*
Links	Graph markieren	Zeichenfläche verschieben
Mitte	Graph kopieren oder entfernen	Operationen rückgängig machen
Rechts	Graph zu- oder aufklappen	Schriftgröße verändern

Tabelle C.12: Zusammenfassung der Maus-Interaktionen

ße von Graphen verändern, entweder indem man einzelne Teilgraphen zu- oder aufklappt oder indem man global die Schriftgröße verändert („pointsize scrolling").

C.3 Implementierungsdetails

Ebenso wie die in Kapitel 4 beschriebene Implementierung von Interaktionsausdrücken, wurde auch der hier vorgestellte Editor in der Programmiersprache `CH` geschrieben. Neben den in Tab. 4.28 (§ 4.8.2) und § A.3.2 genannten Bibliotheksmodulen, wurde eine einfache Graphikbibliothek verwendet, die auf dem X-Window-System [Nye92ab] basiert.[2] Der Quellcode des Programms umfaßt insgesamt etwa 2200 Zeilen.

Ein Graph wird intern, ähnlich wie in § 4.3, als *Operatorbaum* repräsentiert, auf dem sämtliche Änderungsoperationen durchgeführt werden. Allerdings können Sequenz- und Verzweigungsknoten nicht nur zwei, sondern beliebig viele Operanden besitzen.

Jeder Graph bzw. Teilgraph besitzt einen logischen Mittelpunkt sowie Ausdehnungen nach links, rechts, oben und unten, die nach jeder Änderungsoperation mit Hilfe einer rekursiven Funktion `measure()` *von innen nach außen* (engl. bottom up) bestimmt werden. Die Abmessungen von Aktivitäten, Parametern und Faktoren sind hierbei durch die Ausdehnung der entsprechenden Zeichenketten vorgegeben.

Eine zweite rekursive Funktion `position()` legt anschließend *von außen nach innen* (engl. top down) die Koordinaten der Mittelpunkte aller Teilgraphen fest. Die Elemente einer Sequenz werden dabei so ausgerichtet, daß ihre Mittelpunkte auf einer gemeinsamen *horizontalen* Linie liegen, während bei einer Verzweigung die Mittelpunkte aller Zweige auf einer *vertikalen* Linie liegen. Mit diesen Informationen kann eine dritte Funktion `draw()` einen Graphen zeichnen, indem sie zunächst seinen Hintergrund als farbiges Rechteck und anschließend seine geometrischen und textuellen Elemente sowie seine Teilgraphen rekursiv ausgibt. Zur Darstellung benutzerdefinierter Operatoren (wie z. B. der „Blitz-Verzweigung") wird ggf. eine Bitmap-Graphik verwendet.

Eine vierte rekursive Funktion `find()` benutzt ebenfalls die Koordinaten und Ausdehnungen der einzelnen Teilgraphen, um nach jeder Mausbewegung den *aktuellen Graphen* zu bestimmen, d. h. den kleinsten Graphen, der momentan den Mauszeiger enthält und daher invertiert dargestellt wird (vgl. § C.2.2). Außerdem werden die beiden Teilgraphen des aktuellen Graphen ermittelt, *zwischen* denen sich der Mauszeiger gerade befindet, damit die entsprechenden Verbindungslinien rot hervorgehoben werden können (vgl. § C.2.5 und § C.2.6).

Dem üblichen Stil der `CH`-Programmierung folgend, sind alle diese Funktionen als Schar von partiellen Funktionen implementiert (vgl. § 4.4.2.4 und § 4.6.1.2 sowie § A.2.2).

Normalerweise wird ein Graph möglichst kompakt konstruiert, d. h. die horizontalen oder vertikalen Zwischenräume zwischen seinen Teilgraphen werden durch vorgegebene Minimalwerte bestimmt. Um jedoch Verzweigungen mit mehr als zwei Zweigen vollkommen symmetrisch gestalten zu können, muß u. U. zusätzlicher vertikaler Zwischenraum zwischen Zweigen eingefügt werden. So ist im zuvor konstruierten Beispielgraphen der Zwischenraum zwischen den unteren beiden Zweigen des wechselseitigen Ausschlusses etwas größer als zwischen den oberen beiden Zweigen, da andernfalls der vertikale Abstand zwischen ihren Mittelpunkten unterschiedlich und die Struktur der Verzweigung daher asymmetrisch wäre (vgl. z. B. Abb. C.7).

[2] Die Bildschirmausdrucke wurden nichtsdestotrotz unter Microsoft Windows erzeugt.

Anhang D

Wichtige Begriffe und Symbole

Der vorliegende Anhang enthält die folgenden tabellarischen Übersichten:

- Tabelle D.1 faßt die in Kapitel 3 eingeführten Begriffe und Symbole zusammen, die zur Definition der *formalen Semantik* von Interaktionsausdrücken benötigt werden. In der dritten Spalte werden ggf. Variablennamen genannt, mit denen die Elemente der Menge in der ersten Spalte normalerweise bezeichnet werden. Die vierte Spalte gibt jeweils die Nummer des Abschnitts an, in dem das Symbol und der zugehörig Begriff eingeführt werden.
- Tabelle D.2 enthält entsprechend die in § 4.5 und teilweise auch in Anhang B eingeführten Begriffe und Symbole, die zur Definition der *operationalen Semantik* benötigt werden.

Symbol	*Bedeutung*	*Variablen*	*Abschnitt*
Λ Π Ω	Menge aller Aktionsnamen Menge aller Quantorparameter Menge aller konkreten Werte	a_0, b_0 p, q ω, π, ω_i	3.2.1.1
Γ Σ	Menge aller abstrakten Aktionen Menge aller konkreten Aktionen	a, b	3.2.1.2 3.2.1.3
$\langle a_1, \ldots, a_n \rangle$ $\langle\rangle$ $\lvert w \rvert$ Σ^*	Wort (Folge der Aktionen $a_1, \ldots, a_n \in \Sigma$) leeres Wort Länge des Wortes w Menge aller Worte	u, v, w	3.2.2.1
$u\,v$ $U\,V$ U^n U^*	Konkatenation der Worte u und v Konkatenation der Wortmengen U und V n-fache Konkatenation der Wortmenge U mit sich selbst sequentielle Hülle der Wortmenge U		3.2.2.2
$u \otimes v$ $U \otimes V$ $\bigotimes^n U$ $U\#$	Verschränkung der Worte u und v Verschränkung der Wortmengen U und V n-fache Verschränkung der Wortmenge U mit sich selbst parallele Hülle der Wortmenge U		3.2.2.3
$\bigotimes_{i=1}^{n} U_i$ $\bigotimes_{\omega \in \Omega} U_\omega$	Verschränkung endlich vieler Wortmengen $U_1, \ldots, U_n$ Verschränkung unendlich vieler Wortmengen U_ω ($\omega \in \Omega$)		3.2.2.4
w/A	Division des Wortes w durch die Aktionsmenge A		B.2.3.5
Ξ	Menge aller Interaktionsausdrücke	x, y, z	3.3
$\Psi(x)$ $\Phi(x)$	Menge aller partiellen Worte des Ausdrucks x Menge aller vollständigen Worte des Ausdrucks x		3.2.3
$\alpha(x)$ $\kappa_x(y)$	Alphabet des Ausdrucks x Komplement des Alphabets von y bzgl. des Alphabets von x		3.3.1.9 3.3.1.10
a_p^ω x_p^ω	Konkretisierung der Aktion a Konkretisierung des Ausdrucks x		3.3.3.1

Tabelle D.1: Symbole und Begriffe der formalen Semantik

Symbol	*Bedeutung*	*Abschnitt*
$\top$ $\bot$	Wahrheitswert *wahr* Wahrheitswert *falsch*	4.5.1.1
$\vee$ $\wedge$	logische Oder-Verknüpfung logische Und-Verknüpfung	4.5.4.2 4.5.4.4
Θ $\Theta(x)$ $\hat{\Theta}(x)$	Menge aller Zustände Menge aller Zustände des Ausdrucks x Menge aller optimierten Zustände des Ausdrucks x	4.5.1.1 4.5.1.2 4.5.2.3
$\sigma(x)$ $\tau_a(s)$ $\psi(s)$ $\varphi(s)$	initialer Zustand des Ausdrucks x Zustandsübergang des Zustands s durch die Aktion a ist Zustand s ein gültiger Zustand? ist Zustand s ein Endzustand?	4.5.1.1
$\tau_w(s)$ $\psi_w(s)$ $\varphi_w(s)$	Zustandsübergang des Zustands s durch das Wort w ist Zustand $\tau_w(s)$ ein gültiger Zustand? ist Zustand $\tau_w(s)$ ein Endzustand?	4.5.1.2
$\sigma_w(x)$ $\psi_w(x)$ $\varphi_w(x)$	Zustand des Ausdrucks x nach Verarbeitung des Wortes w ist Zustand $\sigma_w(x)$ ein gültiger Zustand? ist Zustand $\sigma_w(x)$ ein Endzustand?	4.5.1.2
$s \cong \hat{s}$ $\rho(s)$	s und $\hat{s}$ sind äquivalente Zustände Optimierung des Zustands s	4.5.2.1 4.5.2.2
$\rho_1,\ \rho_2,\ \rho_3$ $\rho_1',\ \rho_2',\ \rho_3'$	partiell definierte Optimierungsfunktionen vervollständigte Optimierungsfunktionen	4.5.3.2
$\hat{\tau}_a(s)$ $\hat{\tau}_w(s)$ $\hat{\sigma}_w(x)$ $\hat{\psi}_w(x)$ $\hat{\varphi}_w(x)$	optimierter Zustandsübergang des Zustands s durch die Aktion a optimierter Zustandsübergang des Zustands s durch das Wort w optimierter Zustand des Ausdrucks x nach Verarbeitung des Wortes w ist Zustand $\hat{\sigma}_w(x)$ ein gültiger Zustand? ist Zustand $\hat{\sigma}_w(x)$ ein Endzustand?	4.5.2.3
s_p^ω	konkretisierter Zustand	4.5.6.1
$\Omega_a(y,\ p)$ $\Omega_w(y,\ p)$	Menge der relevanten Parameterwerte der Aktion a bzgl. des Ausdrucks y und des Quantorparameters p Menge der relevanten Parameterwerte des Wortes w bzgl. des Ausdrucks y und des Quantorparameters p	4.5.6.2
$\lvert a \rvert$	Breite (Anzahl der Parameter) der Aktion a	4.5.6.2
$[t,\ \omega]$ $\Omega(T)$	erweiterter Zustand (Zustand/Wert-Paar) Menge aller Werte der Zustand/Wert-Paar-Menge T	4.5.6.4
$M_1 \pm M_2$ $M \pm e$	Summe/Differenz der Multimengen M_1 und M_2 Summe/Differenz der Multimenge M und des Elements e	4.5.7.4
$\sum_{i=1}^{n} e_i$ $\sum_{i=1}^{\infty} e$	Multimenge bestehend aus den Elementen $e_1, \ldots, e_n$ Multimenge, die das Element e unendlich oft enthält	4.5.7.4
$\Delta_w^-(y)$ $\Delta_w^{\ominus}(y)$ $\Delta_w^{\circledcirc}$ $\Delta_w^{\circledcirc\!\!\circ}$	Menge der sequentiellen Reste des Wortes w bzgl. des Ausdrucks y Menge der iterativen Reste des Wortes w bzgl. des Ausdrucks y Menge der parallelen Zerlegungen des Wortes w Menge der unendlichen parallelen Zerlegungen des Wortes w	B.2.3.6 B.2.3.7 B.2.3.8 B.2.4.4

Tabelle D.2: Symbole und Begriffe der operationalen Semantik

- Tabelle D.3 faßt die primär in § 3.4.1 und § 3.4.2 definierten *Relationen* zwischen Ausdrücken zusammen.
- Tabelle D.4 enthält in § 4.7 *(Komplexitätsbetrachtungen)* eingeführte und verwendete Begriffe und Symbole.
- Tabelle D.5 gibt einen Überblick über wichtige *Teilklassen* von Interaktionsausdrücken, die zum größten Teil ebenfalls in § 4.7 eingeführt und verwendet werden.
- Die Tabellen D.6 und D.7 schließlich, die logisch eine einzige Tabelle darstellen, geben einen Überblick über sämtliche *Operatoren* von Interaktionsausdrücken bzw. -graphen. Tabelle D.6 zeigt jeweils die anschauliche und formale *Bezeichnung* eines entsprechenden Graphen bzw. Ausdrucks (zusammen mit der Nummer des Abschnitts, in der sie eingeführt wird) sowie die drei äquivalenten Darstellungen als *Graph*, *Ausdruck* und *linearisierter Ausdruck* (vgl. § 4.2.2). Tabelle D.7 enthält einerseits die Definition der *vollständigen* und *partiellen Worte* des Ausdrucks und andererseits Verweise auf Abschnitte, in denen spezielle *Eigenschaften*, die *operationale Semantik* und *Imple-*

Relation	*Bedeutung*	*Abschnitt*
$x_1 \equiv x_2$	x_1 und x_2 sind syntaktisch gleich	3.3.1.9
$x_1 \sim x_2$	x_1 und x_2 sind klassisch äquivalent	3.4.1.1
$x_1 \approx x_2$	x_1 und x_2 sind isoliert äquivalent	
$x_1 = x_2$	x_1 und x_2 sind (umfassend) äquivalent	
$x_1 \subseteq x_2$	x_1 akzeptiert höchstens die Worte, die x_2 akzeptiert	3.4.2.2
$x_1 \leq x_2$	x_1 ist restriktiver als x_2	
$x_1 < x_2$	x_1 ist echt restriktiver als x_2	

Tabelle D.3: Relationen zwischen Ausdrücken

Symbol	*Bedeutung*	*Abschnitt*
$\delta(s)$	Tiefe/Höhe des Zustandsbaums s	4.7.1.1
$\beta(s)$	Verzweigungsgrad des Zustandsbaums s	4.7.1.2
$\gamma(s)$	Größe (Anzahl der Knoten) des Zustandsbaums s	4.7.1.2
$\eta(x)$	Zustandszahl (Anzahl der verschiedenen Zustände) des Ausdrucks x	4.7.1.5

Tabelle D.4: Symbole und Begriffe der Komplexitätsbetrachtungen

Klasse	*Abschnitt*
singuläre Ausdrücke	3.4.6.1
reguläre Ausdrücke	3.5.2
quasi-reguläre Ausdrücke	4.7.2.1
vollständig quantifizierte Ausdrücke	4.7.2.2
homogen quantifizierte Ausdrücke	
injektive Ausdrücke	4.7.2.3
initial fokussierte Ausdrücke	
terminal fokussierte Ausdrücke	
harmlose Ausdrücke	4.7.1.2
gutartige Ausdrücke	
bösartige Ausdrücke	

Tabelle D.5: Teilklassen von Ausdrücken

	Anschauliche und formale Bezeichnung	*Graph*	*Ausdruck*	*Lineare Notation*
Elementare Ausdrücke	Aktion (2.6.1) *atomarer Ausdruck (3.3.1.1)*	a_0 $a_1, \ldots, a_n$	$[a_0, a_1, \ldots, a_n]$	`a0(a1,...,an)`
	Sequenz (2.2.1) *sequentielle Komposition (3.3.1.2)*	y — z	$y - z$	`y - z`
	Wiederholung (2.2.5.1) *sequentielle Iteration (3.3.1.3)*	y	⊖ y	`* y`
	Entweder-oder-Verzweigung (2.2.1) *Disjunktion (3.3.1.4)*	y z	y ○ z	`y | z`
	Eventuell-Verzweigung (2.2.5.2) *Option (3.3.1.5)*	y	◡ y	`? y`
	Sowohl-als-auch-Verzweigung (2.2.2.4) *parallele Komposition (3.3.1.6)*	y z	y ◎ z	`y + z`
	Beliebig-oft-Verzweigung (2.3.1.3) *parallele Iteration (3.3.1.7)*	y	◉ y	`# y`
	Strikte Kopplung (2.3.2.2) *Konjunktion (3.3.1.8)*	y z	y ● z	`y & z`
	Kopplung (2.3.2.2) *Synchronisation (3.3.1.10)*	y z	y ⦿ z	`y @ z`
Multiplikator-Ausdr.	Mehrfach-Ausführung *sequentieller Multiplikator (3.3.2.1)*	n y n	$\overset{n}{⊖}$ y	`-{n} y`
	Mehrfach-Verzweigung (2.2.3.3) *paralleler Multiplikator (3.3.2.1)*	n y n	$\overset{n}{◎}$ y	`+{n} y`
Quantor-Ausdrücke	Für-ein-Verzweigung (2.3.4.6) *Disjunktions-Quantor (3.3.3.1)*	p y p	$\underset{p}{○}$ y	`|[p] y`
	Für-alle-Verzweigung (2.3.4.4) *paralleler Quantor (3.3.3.1)*	p y p	$\underset{p}{◎}$ y	`+[p] y`
	Unendliche Kopplung *Synchronisations-Quantor (3.3.3.1)*	p y p	$\underset{p}{⦿}$ y	`@[p] y`
	Strikte unendliche Kopplung *Konjunktions-Quantor (3.3.3.1)*	p y p	$\underset{p}{●}$ y	`&[p] y`

Tabelle D.6: Operatoren von Interaktionsausdrücken und -graphen (Teil 1)

Vollständige und partielle Worte	*Eigen-schaften*	*Operat. Semantik*	*Implemen-tierung*	*Komplexi-tät*	
$\{\langle a\rangle\} \cap \Sigma^*$ $\{\langle\rangle, \langle a\rangle\} \cap \Sigma^*$		4.5.4.1	4.6.2.1		Elementare Ausdrücke
$\Phi(y)\,\Phi(z)$ $\Psi(y) \cup \Phi(y)\,\Psi(z)$	3.4.7.2	4.5.4.6	4.6.2.3	4.7.3.2	
$\Phi(y)^*$ $\Phi(y)^*\,\Psi(y)$	3.4.8.2 3.4.10.1	4.5.4.7	(4.6.2.3)	4.7.3.2	
$\Phi(y) \cup \Phi(z)$ $\Psi(y) \cup \Psi(z)$	3.4.7.2 3.4.9.1	4.5.4.2	4.6.2.2	4.7.3.1	
$\Phi(y) \cup \{\langle\rangle\}$ $\Psi(y)$	3.4.8.2	4.5.4.3	4.6.2.2	4.7.3.1	
$\Phi(y) \otimes \Phi(z)$ $\Psi(y) \otimes \Psi(z)$	3.4.7.2 3.4.9.1	4.5.4.8	(4.6.2.4)	4.7.3.3	
$\Phi(y)\#$ $\Psi(y)\#$	3.4.8.2 3.4.10.1 3.4.11.1	–	–	4.7.3.3	
$\Phi(y) \cap \Phi(z)$ $\Psi(y) \cap \Psi(z)$	3.4.7.2 3.4.9.1	4.5.4.4	(4.6.2.2)	4.7.3.1	
$\Phi(y) \otimes \kappa_x(y)^* \cap \Phi(z) \otimes \kappa_x(z)^*$ $\Psi(y) \otimes \kappa_x(y)^* \cap \Psi(z) \otimes \kappa_x(z)^*$	3.4.7.2 3.4.9.1	4.5.4.5	(4.6.2.2)	4.7.3.1	
implizit definiert durch Reduktion auf sequentielle Komposition		(4.5.5)		4.7.3.2	Multiplikator-Ausdr.
implizit definiert durch Reduktion auf parallele Komposition		(4.5.5)		4.7.5.2	
$\bigcup_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$ $\bigcup_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	3.4.9.1	4.5.7.1	4.6.4.1	4.7.4.1	Quantor-Ausdrücke
$\bigotimes_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$ $\bigotimes_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	3.4.6.3 3.4.9.1 3.4.11.1	4.5.7.4	4.6.4.2	4.7.4.2	
$\bigcap_{\omega\in\Omega} \Phi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$ $\bigcap_{\omega\in\Omega} \Psi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$	3.4.9.1	4.5.7.3	(4.6.4.1)	4.7.4.1	
$\bigcap_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$ $\bigcap_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	3.4.9.1	4.5.7.2	(4.6.4.1)	4.7.4.1	

Tabelle D.7: Operatoren von Interaktionsausdrücken und -graphen (Teil 2)

mentierung sowie Aussagen zur *Komplexität* des Operators vorgestellt werden.[1] Auf diese Weise hat man für jeden Operator einen „roten Faden“, der sich von der anschaulichen Definition in Kapitel 2 über die formale Behandlung in Kapitel 3 bis zur operationalen Semantik, Implementierung und Komplexität in Kapitel 4 durch die Arbeit zieht.

[1] Die beiden Minuszeichen in der Zeile der parallelen Iteration zeigen an, daß es für diesen Operator keine operationale Semantik und Implementierung gibt (vgl. § 4.3.1), während eingeklammerte Abschnittsnummern darauf hinweisen, daß das entsprechende Thema nur angerissen und nicht vertieft wird. Die Abschnitte 3.4.1 bis 3.4.5, in denen allgemeine Eigenschaften von Interaktionsausdrücken vorgestellt werden, sind nicht aufgeführt.